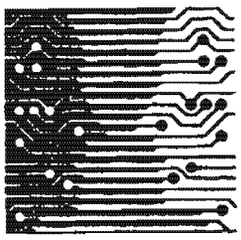
BOTTOM-UP EVALUATION OF HILOG IN THE CONTEXT OF DEDUCTIVE DATABASE SYSTEMS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE,
FACULTY OF SCIENCE
AT THE UNIVERSITY OF CAPE TOWN
IN FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By
Andrew Luppnow
February 1998

Supervised by
P. T. Wood



Abstract

HiLog is a logic-based language which boasts the expressiveness of a higher-order syntax while retaining the simplicity of a first-order semantics. This work examines the suitability of Horn-clause *HiLog* as a query language for deductive databases by investigating the feasibility of adapting well-established Datalog evaluation algorithms for the evaluation of *HiLog* programs. Each of the evaluation algorithms examined in the work is formally described and verified in terms of completeness and correctness. Furthermore, a practical *HiLog* evaluator based on each algorithm verifies the feasibility of its implementation in a real-world context. It is demonstrated that the Datalog evaluation algorithms do indeed have realistic *HiLog* analogs. The work also compares the performance of these analogs.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

One of the major trends in database research today is the development of *deductive database systems.* Such systems attempt to combine data-retrieval and artificial intelligence technologies to provide powerful tools for manipulating and reasoning about data.

A substantial body of the literature on deductive database systems is devoted to evaluation and optimization techniques for programs expressed in Datalog [33], a language based on first-order logic (FOL) and having both a first-order syntax and a first-order semantics. *HiLog* [13] is another language which may serve as a deductive database query language. It offers the expressiveness of a higher-order syntax but, because it retains a first-order semantics, programs written in the language can frequently be evaluated using strategies similar to well-established Datalog evaluation strategies.

This dissertation investigates *HiLog* evaluation algorithms which may all be classified as *bottom-up evaluation strategies.*

1.1 Approach

The work focuses on four *HiLog* evaluation algorithms, each of which may be regarded as an adaptation of one of the following Datalog evaluation algorithms:

- naive evaluation [36, 4, 11, 6],
- seminaive evaluation [6, 5, 9],

- seminaive evaluation benefitting from rule-dependency analysis [8, 23] and
- general seminaive evaluation [30].

In each case the correctness and completeness of the *HiLog* version of the algorithm were established by formal proof. Furthermore, a practical *HiLog* evaluator was implemented to verify the feasibility of the algorithm and to gather data to facilitate comparative performance analyses.

1.2 Overview

The remainder of this document is organized as follows:

Chapter 2 provides a brief overview of the literature pertaining to Datalog, then introduces *HiLog* and formally describes its syntax and semantics. The chapter also examines some of the language's modelling capabilities and presents a few theoretical results which form the basis of later discussions.

Chapter 3 presents an algorithm for performing naive evaluation of a *HiLog* program and proves the algorithm correct and complete. It also describes the proto system, a practical evaluation system based on the naive evaluation algorithm and implemented on a relational database platform.

Chapter 4 presents an algorithm for seminaive evaluation of a *HiLog* program and proves it correct and complete. It also describes the semi system, a modified version of the proto system based on seminaive evaluation. The chapter concludes with a comparison of naive and seminaive evaluation, based on theoretical analyses and the output of the proto and semi systems.

Chapter 5 describes a modified form of the simple seminaive evaluation algorithm which exploits rule dependencies to improve the efficiency of the evaluation. Once again, the algorithm is proved correct and complete. The chapter also describes the sccs system, an evaluator based on the algorithm, and concludes by comparing the performance of the algorithm with that of simple seminaive evaluation.

Chapter 6 describes the *HiLog* analog of the general seminaive (GSN) evaluation algorithm described in [30] and proves the *HiLog* version of the algorithm correct and complete. It also describes the gsn system, an evaluator based on the algorithm, and compares GSN evaluation with ordinary seminaive evaluation.

Chapter 7 presents the conclusions of the work.

Chapter 2

Theoretical Background

This chapter serves primarily to provide a basis for the study of *HiLog* evaluation algorithms. It begins by briefly reviewing the literature on Datalog in Section 2.1 before providing an informal introduction to *HiLog* in Section 2.2. Section 2.3 describes a data model based on *HiLog*, by detailing the syntax and semantics of the language, and presents some essential definitions and theorems to enable a suitably rigorous discussion of *HiLog* evaluation algorithms in succeeding chapters. Section 2.4 compares the *HiLog* and relational data models and concludes that the *HiLog* data model is more expressive. It also examines some modelling features of *HiLog* which make it more convenient than Datalog for reasoning about complex objects. Finally, Section 2.5 identifies a class of *HiLog* programs, based on a subset of a *HiLog* language, which forms the basis of the study described in this thesis.

2.1 An Introduction to Datalog

A number of authors have considered logic as a data model (see, for example, [16, 24]), but, for the purposes of this study, it is most convenient to begin with a description of Datalog [33]. The language readily illustrates how logic may be used to define, reason about and query data and has formed the basis of much research on the evaluation and optimization of logic programs in the context of databases. The reader may find useful overviews of these topics in, for example, [8, 10, 34, 35, 17].

In its purest form, Datalog is simply a Horn clause language [21] which forbids the use of structured terms amongst the arguments of predicate formulas. Traditionally, constants and predicate symbols are represented by disjoint sets of strings comprising only lower-case alphabetic symbols, while variables are represented by upper-case alphabetic symbols. Horn clauses having one or more negative literals are interpreted as rules and written in a Prolog-like syntax.

Consider, for example, the following Datalog program:

$$
\begin{array}{lrcl}
r_1\text{:} & par(henry, sally) & & \\
r_2\text{:} & par(sally, tom) & & \\
r_3\text{:} & anc(X, Y) & :- & par(X, Y) \\
r_4\text{:} & anc(X, Z) & :- & par(X, Y), anc(Y, Z)
\end{array}
$$

The first two clauses have no negative literals and denote *base facts* which may be interpreted as stating that Sally is a parent of Henry and Tom is a parent of Sally. Clauses r_3 and r_4 are rules defining an *anc* (or "ancestor") predicate. Since all the variables of a Horn clause are deemed to be universally quantified, r_3 may be interpreted as: "For all X and Y, if Y is a parent of X, then Y is an ancestor of X." The "," symbol denotes conjunction, so r_4 may be interpreted as "For all X, Y and Z, if Y is a parent of X and Z is an ancestor of Y, then Z is an ancestor of X."

Note that it is possible for a clause to be "unsafe," in that it defines an infinite set of facts. For example, the clause $par(X, Y)$ may be interpreted as "For all X and Y, Y is a parent of X" and clearly defines an infinite relation. A straightforward approach to ensuring safety is to require that every variable which appears in the positive literal of a clause also appears within a negative literal. However, if predicates may include "evaluable" or "built-in" predicates that are interpreted as infinite relations, e.g. $<$, $>$, etc. the conditions for safety require closer examination (see, for example, [37, 34]).

Datalog lends itself to a *model theoretic semantics* in which each predicate can be represented by a database relation, and it is demonstrated in [36] that any pure Datalog program comprising only a finite number of facts and a finite number of safe rules has a finite, least model. Query evaluation strategies based on *bottom-up* evaluation apply the rules of a program to the relations to compute this least model and then extract val-

ues from the model based on the query. The reader is referred to [8] for an overview of goal-driven, *top-down evaluation techniques*, which are not investigated in this thesis.

Bottom-up evaluation techniques include *naive evaluation* [36, 4, 11, 6], *seminaive evaluation* [6, 5, 9] and *general seminaive evaluation* [30] and are investigated closely, in the context of *HiLog* program evaluation, in this work.

A notable extension to Datalog is "Datalog with function symbols" [35], which adds function symbols to the alphabet of the language and permits structured terms to appear as the arguments of predicate formulas, e.g.:

$$real(sum(X, Y)) :- integer(X), real(Y)$$

states that, if X is an integer and Y is a real number, then the sum of X and Y is a real number. *HiLog* makes extensive use of structured terms and issues introduced by such terms, like term-matching, are examined in this work.

A further extension to Datalog introduces "negation," or the ability to write programs in terms of non-Horn clauses. This is addressed in, for example, [32, 3, 25, 28] for Datalog and in [31] for *HiLog*, but does not fall within the scope of this dissertation.

Also worthy of mention, but beyond the scope of this work, are optimization techniques developed for Datalog, including:

- Aho-Ullman [2]
- Kifer-Lozinskii [19]
- Magic Sets [7]
- optimizations for right-, left- and combined linear programs [27]
- factoring optimizations [26] and
- Magic Templates [29]

2.2 An Introduction to *HiLog*

In its full specification [13] *HiLog* allows the contruction of formulas using atomic formulas following the pattern of FOL (see, for example, [21]). However, *HiLog* differs substantially

from FOL and Datalog in the nature of its atomic formulas.

Whereas Datalog requires that the sets of constant symbols, function symbols and predicates of the language's alphabet be disjoint, *HiLog* recognizes only one set of constant symbols, each of which may appear anywhere in a term, so that, for example,

$$a(a, b(a(c)))$$

constitutes a valid *HiLog* term.

HiLog also permits arbitrary terms, including structured terms and terms containing variable symbols, to appear in the functor position of a term. For example,

$$f(X)(f, g)$$

is a valid *HiLog* term.

In *HiLog*, any term constitutes an atomic formula, so the following constitues a valid *HiLog* Horn clause rule:

$$n(X)(b) :- r(X), X(a), c$$

Section 2.3 presents a more complete and formal description of the *HiLog* data model. Section 2.4 compares the expressiveness of *HiLog* with that of relational algebra and describes some of *HiLog*'s modelling abilities.

2.3 The *HiLog* data model

A data model is a formal description of a database which defines the types of the values which may be stored in the database and assigns to the collection of values a semantics in terms of which the answer to a query may be computed. This subsection describes the syntax and semantics of *HiLog*.

2.3.1 Syntax of *HiLog*

The formal syntax of *HiLog* is described in [13] along the following lines:

The alphabet of a *HiLog* language L comprises:

- a countably infinite set V of variables,
- a countable set S of logic symbols,
- punctuation symbols such as ",", "(" and ")",
- the logical connectives "$\wedge$", "$\vee$", "$\neg$", "$\Rightarrow$", "$\Leftarrow$" and "$\Leftrightarrow$" and
- the quantifiers "$\forall$" and "$\exists$".

The inductive definition of a *HiLog* term is as follows:

- if $n \in N$, $n \geq 1$ and, for all i where $i \in N$, $0 \leq i \leq n$, t_i is a term, then $t_0(t_1, \ldots, t_n)$ is a *HiLog* term; t_0 is referred to as the "functor term"; t_1, ..., t_n are referred to as "argument terms";
- a variable is a *HiLog* term;
- a logical symbol is a *HiLog* term.

If a term does not include any variable symbols, it is referred to as a "ground term". The set of all ground terms of a *HiLog* language L is referred to as the "Herbrand universe of L" and is denoted by H_L.

A *HiLog* literal may be defined as follows:

- a term is an atomic formula;
- an atomic formula is a literal (a positive literal);
- the negation of an atomic formula is a literal (a negative literal).

General *HiLog* formulas are constructed from *HiLog* atomic formulas using (,), $\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\Leftarrow$, $\Leftrightarrow$, $\forall$, $\exists$. The rules for constructing the formulas are as for normal predicate calculus.

A *HiLog* clause has the form $\forall X_1 \ldots \forall X_n(L_1 \vee \ldots \vee L_m)$ where the L_1, ..., L_m are *HiLog* literals and the X_1, ..., X_n are all the variables in the literals.

A clause which contains at most one positive literal is referred to as a Horn clause. A Horn clause of the form $A \vee \neg B_1 \vee \ldots \vee \neg B_n$, where A, B_1, ..., B_n are atomic formulas, is a "definite clause". It is generally written using a Prolog-like syntax: $A :\!\!- B_1, \ldots, B_n$.

A finite collection of definite clauses constitutes a *HiLog* program.

2.3.2 Semantics of *HiLog*

The formal semantics of *HiLog* are described in terms of semantic structures as follows [13].

Let L be a *HiLog* language with a set of variables V and a set of logical symbols S.

Semantic Structures

A semantic structure M is a quadruple $(U, U_{true}, I, \mathcal{F})$ where:

- U is a nonempty set of intensions;
- U_{true} is a subset of U; it is the set of intensions associated with those terms which serve as true propositions;
- I is a function defined on S and having values in U; it associates an intension with each logical symbol in S;
- $\mathcal{F}$ is a function defined on U and having values in $\prod_{k=1}^{\infty}[U^k \rightarrow U]$.

Here the "product" denoted by "$\prod$" is the Cartesian product. For any $k \in N$, $k \geq 1$, $[U^k \rightarrow U]$ is the set of all functions defined on U^k and having values in U. It follows that each element of $\prod_{k=1}^{\infty}[U^k \rightarrow U]$ is an infinite tuple $(f_1, \ldots)$ where, for all $i \in N$, $i \geq 1$, f_i is a function defined on U^i and having values in U.

Now let $u \in U$. Let $k \in N$, $k \geq 1$. The kth projection (i.e. kth component) of $\mathcal{F}(u)$ is a function defined on U^k and having values in U. It is denoted by $u_{\mathcal{F}}^{(k)}$.

A variable assignment is a function defined on V and having values in U. It associates an intension with each variable.

A variable assignment ν may be extended to a function defined on the set T of all *HiLog* terms and having values on U. The function ν' is defined as follows:

- for each $X \in V$, $\nu'(X) = \nu(X)$;
- for each $s \in S$, $\nu'(s) = I(s)$;
- for each term t which has the form $t_0(t_1, \ldots, t_n)$, $\nu'(t) = (\nu'(t_0))^n_{\mathcal{F}}(\nu'(t_1), \ldots, \nu'(t_n))$.

Clearly, the extended variable assignment ν' associates an intension with each *HiLog* term in T.

Interpretation of a *HiLog* formula as a proposition

Now let ϕ be an atomic formula of the *HiLog* language L. Let M be a semantic structure for L. Let ν be a variable assignment extended to the term assignment ν'. If $\nu'(\phi) \in U_{true}$ then "ϕ is satisfied by semantic structure M under variable assignment ν" or "M makes ϕ true under variable assignment ν". This is written as "$M \models_\nu \phi$".

In general, for any *HiLog* formula:

- $M \models_\nu (\phi \wedge \psi)$ iff $M \models_\nu \phi$ and $M \models_\nu \psi$;
- $M \models_\nu (\phi \vee \psi)$ iff $M \models_\nu \phi$ or $M \models_\nu psi$;
- $M \models_\nu (\neg\phi)$ iff $M \not\models_\nu \phi$;
- $M \models_\nu ((\forall X)\phi)$ iff, for every variable assignment μ, which may differ from ν on X only, $M \models_\mu \phi$;
- $M \models_\nu ((\exists X)\phi)$ iff, for some variable assignment μ, which may differ from ν on X only, $M \models_\mu \phi$.

Consider a formula ϕ in which every variable appears in the scope of a universal quantifier ($\forall$) or an existensial quantifier ($\exists$). The formula is said to be "closed". Its truth value is clearly independent of any variable assignment and so it is unnecessary to include a subscript after "$\models$". Either $M \models \phi$ or $M \not\models \phi$. Now let P be a *HiLog* program which comprises only definite clauses, each of which is a closed *HiLog* formula. Let ϕ be any other closed *HiLog* formula. If, for any semantic structure M such that M satisfies each clause of P, $M \models \phi$, ϕ is said to be "logically implied by P".

Equality in *HiLog*

HiLog assigns a special semantics to the equality symbol "=". From a syntactic point of view "=" is just another logic symbol and, like any other logic symbol, it has an associated intension, say $u_=$. However, if a semantic structure is to capture the *HiLog* semantics of "=", the infinite tuple of functions, $(f_1, \ldots)$, which $\mathcal{F}$ associates with $u_=$ must satisfy the following requirement:

> for any $k \in N$, $k \geq 1$, and for any $(u_1, \ldots, u_k) \in U^k$, $(u_=)^k_{\mathcal{F}}(u_1, \ldots, u_k) \in U_{true}$ iff $u_1 = \ldots = u_k$.

In other words, the *HiLog* terms in some given set are regarded as equal only if the semantic structure assigns the same intension to all of the terms in the set.

Queries and Answers

The formal semantics of *HiLog* provide a basis for computing the answer to a *HiLog* query. A query is simply a nonground *HiLog* atomic formula. The answer is a relation over a scheme which has one attribute for each of the variables in the atomic formula. Specifically, assume that the query Q has n variables, $X_1, \ldots, X_n$. The answer has relation scheme $(X_1, \ldots, X_n)$ and is a subset of $(H_L)^n$. The tuple t is in the answer relation if and only if the ground term obtained by substituting $t[X_i]$ for X_i in Q (for all i where $1 \leq i \leq n$) is logically implied by the *HiLog* program.

Herbrand Interpretations of a *HiLog* language

A Herbrand Interpretation of a *HiLog* language is essentially a semantic structure which can be described very simply in terms of a subset of the Herbrand Universe of the language.

Let L be a *HiLog* language. Define a semantic structure $\langle U, U_{true}, I, \mathcal{F} \rangle$ for L as follows:

Let $U = H_L$ where H_L is the Herbrand universe of L.

Define I such that, for every logical symbol s in the alphabet of L, $I(s) = s$. This is possible since s is a ground term and thus an element of U.

Define $\mathcal{F}$ such that for every $u \in U$, $\mathcal{F}(u)$ is an infinite tuple of functions $(f_1, \ldots)$ satifying the following: for all $i \in N$, $i \geq 1$, and for all $(u_1, \ldots, u_i) \in U^i$, $f_i(u_1, \ldots, u_i)$ is the ground term $u(u_1, \ldots, u_i)$. Note that $u(u_1, \ldots, u_i)$ must be a ground term, and thus an element of U, since $u, u_1, \ldots, u_i$ are all elements of U and therefore ground terms.

The resulting semantic structure is referred to as a *Herbrand interpretation.*

Note that, for a Herbrand interpretation, it is not necessary to specify U, I or $\mathcal{F}$ since these can be derived from the language L. It is sufficient to specify U_{true}. This is just the set of all *HiLog* ground terms which the interpretation regards as true propositions.

Now it is easy to see that, in the context of a Herbrand Interpretation, a variable assignment ν associates *HiLog* ground terms with variables, and its corresponding "extended variable assignment" ν' is simply a function which, when applied to a term t, systematically and simultaneously replaces the variables of t with those ground terms. This leads to the following working definitions of *variable assignment* and *substitution*:

Definition 1 (Variable Assignment) *Let L be a language of HiLog with alphabet A. Let V be the set of all variable symbols in A. Let S be the set of all "constant symbols" (logic symbols) in A. Observe that $V \cap S = \emptyset$. Let H_L be the Herbrand Universe of L, i.e. H_L is the set of all HiLog ground terms t s.t. every constant symbol in t is an element of S.*

A variable assignment ν is a subset of $V \times H_L$, i.e. it is a set of ordered pairs, each of the form (v, t) where $v \in V$ and $t \in H_L$. Furthermore, if (v_1, t_1) and (v_2, t_2) are any two distinct elements of ν, then $v_1 \neq v_2$.

If $v \in V$ and $(v, t) \in \nu$, then say "v is bound *under ν" and "ν binds t to v". If v is bound under ν, the ground term which ν binds to v is denoted by $\nu(v)$. If $v \in V$ and ν contains no pair (v, t) where $t \in H_L$, then say "v is unbound under ν".*

Definition 2 (Substitution) *Let t be a HiLog term which may contain variable symbols. Let ν be a variable assignment under which each variable in t is bound. The HiLog ground term obtained by simultaneously substituting $\nu(v)$ for every variable v in t is denoted by $t\nu$.*

It is also worth noting that a *HiLog* formula which may be expressed as a conjunction of definite clauses may be regarded as specifying a set F of facts and a set P of rules and that the formula will only be satisfied by a Herbrand Interpretation which contains all the ground terms of F and satisfies each of the rules in P. This leads to the notion of a *model.*

Definition 3 (Herbrand Model) *The* Herbrand Model *of a program P and a set of facts F is a (not necessarily proper) superset of F which satisfies every rule in P.*

In applying bottom-up evaluation to a set of facts and rules, it is desirable to avoid the computation of any extraneous facts. Thus the objectives of the evaluation algorithms in succeeding chapters will be stated in terms of *least models.*

Definition 4 (Least Model) *A model M is a* least model *iff, for every Herbrand interpretation I s.t. I is a model, $M \subseteq I$.*

It remains to prove a number of important properties of least models to facilitate the formal analyses of evaluation algorithms in succeeding chapters. Note that the FOL counterparts of these theorems are discussed in [21].

Theorem 1 *A least model of a program P and a set of facts F is unique, so that one may speak of* the *least model.*

Proof: Assume the assertion is false, then there exists at least one pair of least models M_1 and M_2 s.t. $M_1 \neq M_2$.

Since M_1 is a least model and M_2 is a model, $M_1 \subseteq M_2$ by the definition of a least model. Similarly, since M_2 is a least model and M_1 is a model, $M_2 \subseteq M_1$ by the definition of a least model. Thus $M_1 = M_2$, which contradicts the assumption that the theorem is false. □

Theorem 2 *Let P be a program comprising only definite HiLog Horn clauses. Let F be a set of HiLog facts. Let M_1 and M_2 both be models of P and F. Then $M_1 \cap M_2$ is a model of P and F.*

Proof: First note that, since M_1 and M_2 are both models, M_1 and M_2 are both supersets of F. Thus $M_1 \cap M_2$ is a superset of F.

Now assume that $M_1 \cap M_2$ is not a model. Then there exists a Horn clause C in P and a variable assignment ν s.t. $C\nu$ is not satisfied by $M_1 \cap M_2$. Specifically, if $C = A_0 \vee \neg A_1 \vee \ldots \vee \neg A_n$ where $n \in N$, $n \geq 1$ and $A_0, \ldots, A_n$ are *HiLog* atomic formulas, $A_0\nu \notin M_1 \cap M_2$ and $A_1\nu, \ldots, A_n\nu$ are all elements of $M_1 \cap M_2$. Thus $A_1\nu, \ldots, A_n\nu$ are all elements of M_1 and, since M_1 is a model, M_1 must satisfy $C\nu$, and so $A_0\nu \in M_1$. Similarly, $A_1\nu, \ldots, A_n\nu$ are all elements of M_2 and, since M_2 is a model, M_2 must satisfy $C\nu$, and so $A_0\nu \in M_2$. It follows that $A_0\nu$ is indeed an element of $M_1 \cap M_2$ and the resulting contradiction forces the conclusion that $M_1 \cap M_2$ is a model. □

Theorem 3 *Let P be a HiLog program comprising only definite Horn clauses and let F be a set of HiLog facts. Let $M(P,F)$ be the set of all Herbrand interpretations which are models of P and F. Let $\bigcap M(P,F)$ denote the Herbrand interpretation which is the intersection of all elements of $M(P,F)$. Then $\bigcap M(P,F)$ is the least Herbrand model of P and F.*

Proof: By Theorem 2, $\bigcap M(P,F)$ is clearly a model of P and F. Furthermore, every ground term α in $\bigcap M(P,F)$ must appear in every element of $M(P,F)$, otherwise it would not be an element of $\bigcap M(P,F)$. It follows that, for every I in $M(P,F)$, $\bigcap M(P,F) \subseteq I$. In other words, $\bigcap M(P,F)$ is a model which is a subset of every other model and, by the definition of least model, $\bigcap M(P,F)$ is thus the least Herbrand model of P and F. □

2.4 Comparing *HiLog* and relational algebra

2.4.1 Modelling relational algebra with *HiLog*

Several texts containing introductory discussions on Datalog, e.g. [34, 17], compare the expressive power of Datalog and relational algebra and describe the techniques for modelling relations and relational algebra expressions with Datalog programs. Since the Datalog programs generated by those techniques are also *HiLog* programs, this section presents only a brief overview of the topic.

Recall that the relational data model describes a database in terms of a collection of relations, each of which is defined over a relation scheme. A relation scheme, in turn, is specified as a list of attributes, each of which has a domain of atomic elements. Thus a relation scheme comprising n attributes ($n \in N$, $n \geq 1$), may be represented as $(a_1, \ldots, a_n)$ where, for each $i \in N$ and $1 \leq i \leq n$, a_i is an attribute with domain D_i. A relation over this scheme is simply a subset of $D_1 \times \cdots \times D_n$. It is a set of tuples $(v_1, \ldots, v_n)$ where, for each $i \in N$ and $1 \leq i \leq n$, $v_i \in D_i$.

The relational model may assign to a database a declarative semantics (captured by the relational calculus) or a procedural semantics (captured by the relational algebra). Since the algebra and the calculus are equivalent, it is sufficient to consider only the algebra.

Recall that relational algebra provides five fundamental relational operators which may be applied to relations to generate the answers to queries. The operators are *select* (σ), *project* (π), *Cartesian product* ($\times$), *union* ($\cup$) and *difference* ($-$).

To show that the *HiLog* data model is at least as powerful as the relational data model it is necessary to show:

- that *HiLog* facts can model relations and
- that *HiLog* rules can model each of the five fundamental relational operators.

A relation over a relation scheme comprising n attributes may be represented by a set of *HiLog* facts, each having n arguments. For each tuple of the relation, the set includes one fact whose functor term is the name of the relation and whose n arguments are the n components of the tuple.

The modelling of the relational operators is most easily demonstrated by example. Assume that r and s are defined over a relation scheme comprising three attributes and that t is defined over a relation scheme comprising two attributes. Note also that the notation $\$i$ is used to denote the ith attribute. The following sub-sections show how relational algebra expressions over r, s and t can be encoded in *HiLog*.

Select operator

The expression $\sigma_{\$2=c}(r)$, where c is a constant, can be encoded as:

$$r_1\colon \quad answer(X_1, c, X_3) \quad :- \quad r(X_1, c, X_3).$$

The expression $\sigma_{\$1=\$3}(r)$ can be encoded as:

$$r_1\colon \quad answer(X_1, X_2, X_1) \quad :- \quad r(X_1, X_2, X_1).$$

Note the use of logic symbols and repeated variables to express the selection constraints.

Project operator

The expression $\pi_{\$1,\$2}(r)$ can be encoded as

$$r_1\colon \quad answer(X_1, X_2) \quad :- \quad r(X_1, X_2, X_3).$$

The argument list of the head literal is just the appropriate sublist of the body literal's argument list.

Cartesian product operator

The expression $r \times t$ can be encoded very simply using conjunction in the body of the rule to combine the tuples of the two relations:

$$r_1\colon \quad answer(X_1, X_2, X_3, Y_1, Y_2) \quad :- \quad r(X_1, X_2, X_3), t(Y_1, Y_2).$$

Observe that the join operator, expressed in terms of Cartesian product, selection and projection in relational algebra, has a succinct encoding in *HiLog*. For example, if r and t are joined on one attribute, which is the third attribute of r and the first attribute of t, then the encoding is:

$$r_1\colon \quad answer(X_1, X_2, X_3, Y_1) \quad :- \quad r(X_1, X_2, X_3), t(X_3, Y_1).$$

Union operator

The expression $r \cup s$ is easily encoded using two rules; one rule provides the tuples of r, the other provides the tuples of s:

$$r_1: \quad answer(X_1, X_2, X_3) \quad :- \quad r(X_1, X_2, X_3).$$
$$r_2: \quad answer(X_1, X_2, X_3) \quad :- \quad s(X_1, X_2, X_3).$$

Difference operator

The expression $r - s$ may be encoded using negation to exclude the tuples of the s relation:

$$r_1: \quad answer(X_1, X_2, X_3) \quad :- \quad r(X_1, X_2, X_3), \neg s(X_1, X_2, X_3).$$

Note that intersection, which may be expressed in terms of difference in relational algebra ($r \cap s = r - (r - s)$), has a straightforward encoding in *HiLog*. For example, the intersection of r and s is provided by:

$$r_1: \quad answer(X_1, X_2, X_3) \quad :- \quad r(X_1, X_2, X_3), s(X_1, X_2, X_3).$$

2.4.2 Other modelling abilities of *HiLog*

Since relations and all the fundamental relational operators can be modelled in *HiLog*, the language is clearly at least as powerful as relational algebra. The *HiLog* data model also has several features which make it significantly more powerful than the relational data model. This section briefly examines some of those features. A more comprehensive discussion of the abilities of *HiLog* can be found in [13].

Structured Terms

The description of the relational model in the previous subsection emphasizes that the domains of attributes may contain only *atomic* elements. Thus a tuple may be regarded

as a fixed-length list of atomic values. Note, however, that the arguments of a *HiLog* atomic formula may have arbitrarily complex structures. Thus *HiLog* facts can model relations whose tuples comprise structured elements. In principle, a database based on the relational model can be used to stored such structured terms if they are mapped onto atomic elements, but the relational model does not provide operations for accessing the subterms of a structured term. So a rule such as

$$r_1: \quad p(X) \quad :- \quad q(f(X)).$$

is possible under the *HiLog* model, but it has no equivalent under the relational model.

Recursion

A very important feature of the *HiLog* data model is its ability to support the semantics of *recursive* rules. For example, assume that r is a binary relation. The transitive closure of r may be defined using the following two *HiLog* rules:

$$r_1: \quad rclosure(X,Y) \quad :- \quad r(X,Y).$$
$$r_2: \quad rclosure(X,Y) \quad :- \quad r(X,Z), rclosure(Z,Y).$$

Note that the second rule defines $rclosure$ in terms of itself.

Now let M be any model which interprets r as a set of true facts of the form $r(x_1, x_2)$ and which satisfies the two rules. It can be proved by induction that if r contains a sequence of binary tuples $(x_1, x_2), (x_2, x_3), \ldots, (x_{n-2}, x_{n-1}), (x_{n-1}, x_n)$, where the second component of each tuple is equal to the first component of the next tuple, then $rclosure(x_1, x_n)$ must be satisfied by M.

Note that the application of the relational algebra assignments

$$rclosure \quad := \quad r$$
$$rclosure \quad := \quad r \bowtie_{\$2=\$1} rclosure$$

will generate a "closure" which recognizes only sequences of length two. To generate the complete closure it is necessary to apply the assignments repeatedly until no new tuples are added to $rclosure$. The relational model, however, provides only the fundamental operators of select, project, Cartesian product, union and difference. It does not provide constructs for looping or for conditional execution.

Complex Functor Terms

HiLog supports terms with *complex functor terms.* This means that it supports atomic formulas in which the "predicate" may be an arbitrary term. The ability to use such atomic formulas allows the definition of *generic predicates.* For example, a generic *closure* predicate can be defined using the following two *HiLog* rules:

$$r_1\colon\ closure(R)(X,Y) \ :-\ R(X,Y).$$
$$r_2\colon\ closure(R)(X,Y) \ :-\ R(X,Z), closure(R)(Z,Y).$$

These rules allow a database client to compute the closure of *any* binary relation without having to write a separate program for each relation. So the closure of, say, relation p could be obtained with the query goal $?{-}closure(p)(X,Y)$.

Terms with multiple roles

HiLog allows an arbitrary term to assume the role of a functor, an argument, a predicate or even an entire atomic formula. An application of this might be to define a set and use that set as a component of a structured object. For example:

$$children(bob)(sally).$$
$$children(bob)(timmy).$$

defines a set $children(bob)$ which contains the two elements *sally* and *timmy.* Now assert the fact:

$$employee(bob, sales, children(bob)).$$

This declares that Bob is an employee of the sales department whose children belong to the set $children(bob)$. Consider a rule for describing the set of all children of all employees of the sales department. The following suffices:

$$r_1\colon\ childset(X) \ :-\ employee(A, sales, C), C(X).$$

The *childset* set must clearly include *sally* and *timmy.*

Note how the term $children(bob)$ assumes the role of both a functor term and an argument term. Similarly, the rule r_1 includes two occurrences of the variable C—in the first body literal it appears as an argument and in the second it appears as a functor.

2.5 A subset of *HiLog* for querying databases

One of the objectives of this thesis is to develop experimental deductive database systems which use *HiLog* to define and query data. These systems are based on a subset of *HiLog* which is defined in this section.

An initial restriction requires that a *HiLog* program contain only Horn clauses. Thus each formula of the program is either a fact or a rule of the form $A :- B_1, \ldots, B_n$, where A and the B_i are all atomic formulas. The thesis does not address the evaluation of *HiLog* programs whose rule bodies include negative literals.

If the database system is to be based on Herbrand interpretations, a further restriction should apply. It is necessary to forbid the use of the "equality" logic symbol (=) in a fact or rule head. To see why, assume that a program asserts the fact $= (t_1, t_2)$, where t_1 and t_2 are distinct ground terms. A Herbrand interpretation for the program must include $= (t_1, t_2)$. But t_1 and t_2 have different intensions under a Herbrand interpretation. Thus, if the Herbrand interpretation is to capture the semantics of *HiLog* equality, it cannot include $= (t_1, t_2)$. It may be argued that, for any semantic structure M, it is possible to construct an equivalent Herbrand interpretation containing precisely those ground terms which are satisfied by M. However, this means that, whenever a single fact is asserted, a potentially very large number of ground terms must be added to the Herbrand interpretation to ensure that *HiLog* equality is accurately simulated. It thus seems reasonable to begin by applying the suggested restriction to the use of "=".

Finally, the chosen subset of *HiLog* excludes rules whose heads or bodies contain ground literals. It is straightforward to extend the algorithms described in this thesis to support such rules using techniques similar to those employed by analogous Datalog algorithms [34] to support Datalog rules that include ground literals. Nonetheless, the restriction is imposed because it significantly simplifies the theoretical analyses of the algorithms.

Chapter 3

Naive Evaluation

This chapter describes the proto system, a somewhat primitive "prototype" *HiLog* evaluator which was designed to meet the following objectives:

- to enable an examination of *HiLog* as a language for defining data, reasoning about data and querying data in a deductive database system,
- to investigate the application of the naive evaluation algorithm (described for Datalog in [34, 35, 6]) to the bottom-up evaluation of *HiLog*,
- to highlight the inefficiencies of the naive evaluation algorithm and
- to provide a basis for the development of more efficient *HiLog* evaluators.

The chapter begins by describing an intuitive approach to the problem of finding the least Herbrand model of a given set of *HiLog* facts and rules. Section 3.2 provides a more rigorous discussion of the ideas introduced here and presents algorithms for the important tasks of term-matching and rule application, as well as an algorithm for naive evaluation itself. Finally, the chapter concludes with a brief overview of the design of the proto system.

3.1 An Overview of Naive Evaluation

This section presents an informal overview of the naive evaluation algorithm. It forms the basis of a more complete discussion of the topic in the succeeding sections of the chapter.

The objective of naive evaluation is to compute the least Herbrand model of a a set of *HiLog* facts and a *HiLog* program. This entails finding the least set of facts which is a superset of the given set of facts and which satisfies all the rules of the program.

First consider the process of augmenting a set of *HiLog* facts so that the new set satisfies a *single HiLog* rule. Specifically, let c be the simple rule $A_0 :- A_1$, where A_0 and A_1 are nonground *HiLog* terms, and let I be a given set of *HiLog* facts. Now assume that it is necessary to add facts to I to ensure that I satisfies c.

Recall from the previous chapter that c may be denoted by the *HiLog* formula $A_0 \vee \neg A_1$ and that, if I is to satisfy c, it must make the formula true under every assignment of ground terms to the variables of c. Now, if A_1 is false under a variable assignment ν, then the formula is clearly true under ν, but if A_1 is true under ν, A_0 must also be true under ν if the formula is to be satisfied. This suggests that it is possible to use the following procedure to compute the set of facts which must be added to I: for each fact t *presently* in I, establish whether or not there exists a variable assignment ν s.t. $A_1\nu = t$; if so, add $A_0\nu$ to I.

Example 1 Let c be the *HiLog* rule $p(X, Y, f(X)) :- p(a, X, f(Y))$ and let I be the set of facts $\{p(b, a, f(a)), p(a, a, f(b)), p(a, b, g(h))\}$. Here it is necessary to consider each fact in I and establish whether or not there exists a variable assignment which makes the subgoal $p(a, X, f(Y))$ identical to the fact. Clearly, there is no such variable assignment for $p(b, a, f(a))$, since the subgoal has a as its first argument, while the fact has b as its first argument. However, the variable assignment $\{(X, a), (Y, b)\}$ makes the subgoal identical to the second fact, $p(a, a, f(b))$. Thus the fact obtained by substituting a for X and b for Y in the head of the rule, i.e. $p(a, b, f(a))$, may be added to I. Now consider the third fact, $p(a, b, g(h))$. Since the third argument of the subgoal has f as its functor term, while the third argument of the fact has g as its functor term, no variable assignment can possibly make the subgoal identical to the fact, and so no further facts are added to I. □

Given a nonground *HiLog* term t and a *HiLog* ground term t', the process of finding a variable assignment ν s.t. $t\nu = t'$ (assuming one exists) is referred to as *term-matching.* Section 3.2.1 of this chapter discusses term-matching in greater detail and presents an algorithm for the operation.

The *rule application* procedure involves finding variable assignments under which a given set of *HiLog* ground terms satisfies the body of a rule, substituting for the variables in the rule head, and then adding the resulting ground terms to the set. Example 1 illustrated how term-matching could be used to perform this operation for a rule having only one subgoal in the rule body. If the rule body contains two or more subgoals, the procedure is slightly more complicated, albeit still based on term-matching. Section 3.2.2 of this chapter examines rule application more closely and describes an algorithm which successfully deals with the general case.

Now, while term-matching and rule application are certainly essential steps in the naive evaluation algorithm, they are not, by themselves, adequate for computing a model for *any* given program. To see why, refer once more to Example 1. There it was noted that the application of the rule to the original set I of *HiLog* facts yielded a new fact, $p(a, b, f(a))$, which had to be added to I. But now the variable assignment $\{(X, b), (Y, a)\}$ makes the subgoal, $p(a, X, f(Y))$, identical to the new fact and substituting b for X and a for Y in the rule head yields yet another new fact, $p(b, a, f(b))$. This fact must also be added to I if I is to satisfy the rule. At this stage it would appear that no new facts can be generated, since the latest addition to I cannot match the subgoal of the rule. The reader may verify that the final set of facts, $\{p(b, a, f(a)), p(a, a, f(b)), p(a, b, g(h)), p(a, b, f(a)), p(b, a, f(b))\}$, does indeed satisfy the rule.

This discussion illustrates that, if the application of a rule to a set of facts can produce new facts which, in turn, match a subgoal of the rule, a single application of the rule is generally not sufficient. Rather, it is necessary to repeatedly apply the rule to a growing set of facts until no new facts are generated. The following example demonstrates that this approach can easily be extended to deal with programs which comprise more than one rule.

Example 2 Let P be a *HiLog* program defined in terms of the two rules $p(X, Y) :- q(X, Y)$

and $q(Y,X)$:- $p(X,Y)$, and let I be a set containing the single *HiLog* fact $q(a,b)$. An application of the first rule adds a new fact, $p(a,b)$, to I. Then an application of the second rule adds another new fact, $q(b,a)$, to I. But now observe that $q(b,a)$ matches the subgoal of the first rule, so that, if I is to satisfy both rules of the program, the first rule must be applied again to add the fact $p(b,a)$ to I. Since another application of the second rule yields no new facts, it is reasonable to assume that the final set of facts, $\{q(a,b), p(a,b), q(b,a), p(b,a)\}$, satisfies the program, and this is indeed the case. □

Example 2 suggests a straightforward solution to the problem of computing a model of a given set of *HiLog* facts and a given *HiLog* program: repeatedly apply all the rules of the program to the set of facts until no rule application generates any new facts. This simple procedure is known as *naive evaluation* and it forms the basis of the bottom-up evaluation algorithm detailed in this chapter. Section 3.2.3 proves that, if naive evaluation is applied to a finite set of facts and a program comprising only Horn clause rules, and if the facts and the program have a *finite* least Herbrand model, then the procedure will duly compute that model.

3.2 Algorithms

3.2.1 The Term-matching Algorithm

Term-matching is an important and fundamental operation in any deductive database system which is required to manipulate recursively-structured terms. In fact, it may be regarded as a generalized form of relational algebra's *select* operation, in that it is capable of dealing with "structured tuples" in addition to ordinary "flat tuples." This subsection describes the operation, presents an algorithm for performing the operation and proves the completeness and correctness of the algorithm. Refer to [35] for a discussion of term-matching in the context of Datalog.

Recall from the informal introduction to term-matching in Section 3.1 that the purpose of the operation may be stated as follows: given a *HiLog* term t, which may contain variable symbols, and a *HiLog* ground term t', establish whether or not there exists a variable assignment ν s.t. $t\nu = t'$ and, if so, find ν.

The definition of *HiLog* terms in the previous chapter stresses their recursive nature, so it should not be surprising that the most convenient algorithm for term-matching is a recursive one. Given a structured term t, say $t_0(t_1, \ldots, t_n)$, and a ground term t', the algorithm first establishes whether or not t' is also a structured term comprising $(n+1)$ subterms. If not, it immediately returns FALSE. However, if t' *is* a structured term comprising $(n+1)$ subterms, say $t'_0(t'_1, \ldots, t'_n)$, the algorithm invokes itself recursively to establish whether each t'_i can match its corresponding t_i. Now observe that it is generally not sufficient to test these pairs of subterms independently, since it may be that matching one pair of subterms requires a variable assignment ν_1, while matching a second pair of subterms another variable assignment, ν_2, which is in conflict with ν_1. Example 3 below demonstrates how this situation may arise.

Example 3 Let t be the *HiLog* term $p(f(X), X)$ and let t' be the *HiLog* ground term $p(f(a), b)$. Note that both terms have the same structure and that corresponding terms can easily be made to match: the functor terms match because they are both just the constant p; the first arguments can be made to match by binding a to X; the second arguments can be made to match by binding b to X. However, because it is necessary to bind different values to X in order to match the first and second pairs of arguments, no valid variable assignment can make $p(f(X), X)$ identical to $p(f(a), b)$. □

The algorithm described in this section adopts a simple, but effective, approach to dealing with such conflicts. It maintains a global variable assignment ν and refers to the assignment whenever it needs to match a variable, say v, and a ground term, say t'. If the variable is unbound under ν, the algorithm just adds (v, t') to ν and returns TRUE. But if v is already bound under ν, say $\nu(v) = t''$, the algorithm checks whether t' is identical to t'', returning TRUE if it is, FALSE if it is not.

Example 4 Let t be the *HiLog* term $X(Y, X)$, let t' be the *HiLog* ground term $a(b, a)$ and let ν be an initially empty global variable assignment. Since the terms are both structured terms comprising three subterms, the algorithm does not terminate immediately, but invokes itself recursively for each pair of corresponding subterms. The first recursive invocation notes that X is unbound under ν, so it adds (X, a) to ν and returns TRUE.

Similarly, the second recursive invocation just adds (Y, b) to ν and returns TRUE. However, the third recursive invocation notes that ν already binds the ground term a to X, so it compares this ground term with the third subterm of t' and, finding that they are identical, returns TRUE. Finally, because each recursive invocation has returned TRUE, the original invocation returns TRUE, leaving ν equal to $\{(X, a), (Y, b)\}$. □

To facilitate a more formal description of the term-matching algorithm, it is necessary to introduce the notion of a *restricted assignment.*

Definition 5 (Restricted Assignment) *Let ν be a variable assignment. Let t be a HiLog term containing all the variable symbols in a set W of variable symbols, and only those variable symbols. The variable assignment "ν restricted to the variables of t", denoted by ν_t, is the set of all ordered pairs (v, t) s.t. $(v, t) \in \nu$ and $v \in W$.* □

Example 5 If t is the *HiLog* term $p(X)(Y, g(X))$ and ν is the variable assignment

$$\{(W, a), (X, b), (Y, c), (Z, d)\}$$

then ν_t is just $\{(X, b), (Y, c)\}$. □

The algorithm for term-matching is based on the recursive procedure match, detailed in Figure 3.1 and may be regarded as a generalization of the term-matching algorithm described in [35] in the context of Datalog. The procedure accepts two arguments t and t' and operates in the presence of a global variable assignment ν. It returns the boolean value TRUE iff there exists a variable assignment σ over the variables of t s.t. $t\sigma = t'$ and ν_t (i.e. ν restricted to the variables in t) is a subset of σ; otherwise it returns FALSE. Furthermore, if match returns TRUE it updates ν so that ν_t is equal to σ.

The theorems that follow prove the completeness and correctness of the match procedure. They rely on the notion of "term height" as defined below.

Definition 6 (Height of a Term) *Define the height of a HiLog term t, denoted by $h(t)$, as follows:*

$$h(t) = \begin{cases} \max(h(t_0), \ldots, h(t_n)) + 1 & \text{if } t \text{ is of the form } t_0(t_1, \ldots, t_n) \\ 1 & \text{if } t \text{ is a variable or a constant} \end{cases}$$

□

```
1:  boolean match(t, t')
    /* t is a HiLog term which may include variables; t' is a HiLog ground
    term; ν is a global variable assignment. */
2:  {
3:      if (t has the form t_0(t_1, ..., t_n))
4:          if (!(t' has the form t'_0(t'_1, ..., t'_n)))
5:              return(FALSE);
6:          else
            /* t' has the form t'_0(t'_1, ..., t'_n) */
7:          {
8:              for (i = 0; i <= n; i++)
9:                  if (!match(t, t'))
10:                     return(FALSE);
                /* For every i ∈ N, 0 ≤ i ≤ n, match(t_i, t'_i)
                returned TRUE. */
11:             return(TRUE);
12:         }
13:     else
14:         if (t is a variable v)
15:             if (v is bound under ν)
                    /* t is a variable bound under ν */
16:                 if (ν(v) == t')
17:                     return(TRUE);
18:                 else
19:                     return(FALSE);
20:             else
                /* t is a variable which is not bound under ν */
21:             {
22:                 ν = ν ∪ {(v, t')};
23:                 return(TRUE);
24:             }
25:         else
                /* t is a constant */
26:             if ((t' is a constant) && (t' == t))
27:                 return(TRUE);
28:             else
29:                 return(FALSE);
30: }
```

Figure 3.1: Recursive procedure for term-matching

Example 6 The terms X and a each have a height of 1. The term $p(Y)$ clearly has a height of 2. The term $p(Y)(X, a)$, formed by combining the first three terms, has a height of 3 because its "highest subterm," $p(Y)$, has a height of 2. □

Theorem 4 *Let t be a HiLog term which may include variable symbols; let t' be a HiLog ground term; assume that there exists a variable assignment σ over the variables of t s.t. $t\sigma = t'$ and let ν be a variable assignment s.t. ν_t (i.e. ν restricted to the variables in t) is a subset of σ. Then* match(t, t') *returns* TRUE *and updates ν so that ν_t is equal to σ.*

Proof: The proof is an induction on the height of the argument t. Assume that the theorem holds for all $h(t)$ s.t. $1 \leq h(t) \leq m$. Now assume that $h(t) = m + 1$. Since $h(t)$ is then greater than one, t cannot be a variable or a constant—it must be complex term of the form $t_0(t_1, \ldots, t_n)$, where $n \in N$, $n \geq 1$ and for all $i \in N$, $0 \leq i \leq n$, t_i is a *HiLog* term with $h(t_i) \leq m$. Also, if t' matches t, then t' must be of the form $t'_0(t'_1, \ldots, t'_n)$. Thus the compound statement of lines 7–12 will be executed. For each $i \in N$, $0 \leq i \leq n$, let $\sigma_i = \sigma_{t_i}$ (i.e. σ restricted to the variables of t_i) and note that, since $t\sigma = t'$, $t_i\sigma_i$ must be equal to t'_i. Also, since ν_t is a subset of σ, ν_{t_i} must be a subset of σ_i. Thus it follows from the inductive hypothesis that, for any $i \in N$, $0 \leq i \leq n$, a call to match(t_i, t'_i) will return TRUE. Furthermore, the call updates ν so that $\nu_{t_i} = \sigma_i$ and so, since the call can only add a binding (v, g) to ν if v is a variable in t_i, and since a variable of t_i is clearly a variable of t, ν_t must *remain* a subset of σ. Thus it follows that *all* the recursive calls to match in the for-statement of lines 8–10 will return TRUE and so match(t, t') will duly return TRUE when line 11 is executed. To complete the proof of the inductive step, it suffices to show that when match(t, t') returns, ν_t is equal to σ. It has already been argued that ν_t remains a subset of σ after each recursive call to match in the for-statement. It follows that, when line 11 is executed, ν_t is a subset of σ. Now assume that, for some variable v and some *HiLog* ground term g, $(v, g) \in \sigma$. Then, for some $i \in N$, $0 \leq i \leq n$, t_i contains the variable v and so $(v, g) \in \sigma_i$. By the inductive hypothesis, the call to match(t_i, t'_i) adds (v, g) to ν (if it is not already there) and so $(v, g) \in \nu_t$. It follows that, when line 11 is executed, σ is a subset of ν_t and, since ν_t is also a subset of σ, $\nu_t = \sigma$.

For the basis of the induction, i.e. $h(t) = 1$, observe that t must be a variable or a constant. If t is a variable, say v, then $\sigma = \{(v, t')\}$ and since $\nu_t \subseteq \sigma$, $\nu_t = \{(v, t')\}$ or $\nu_t = \emptyset$. In

the former case $\nu(v) = t'$ and the conditions on lines 14, 15 and 16 all test true, so line 17 will be executed. In the latter case the condition of line 15 tests false and the compound statement of lines 21–24 is executed, adding (v, t') to ν so that $\nu_t = \{(v, t')\}$. In either case match(t, t') clearly returns TRUE and leaves ν_t equal to σ. If t is a constant and t' matches t, then clearly t' is also a constant and $t = t'$. So the conditions on line 26 both test true and match(t, t') returns TRUE. Also, since t contains no variable symbols, $\nu_t = \sigma = \emptyset$. □

Theorem 5 *If* match(t, t') *returns* TRUE *then*

1. *there exists a valid variable assignment σ over the variables of t s.t. $t\sigma = t'$ and*
2. *prior to execution of* match(t, t'), *ν_t (i.e. ν restricted to the variables of t) was a subset of σ.*

Proof: The proof is an induction on the height $h(t)$ of the argument t. Assume that the theorem holds for any $h(t)$ s.t. $1 \leq h(t) \leq m$. Now assume that a call to match(t, t') in the presence of the global variable assignment ν returns TRUE and that $h(t) = m + 1$. Since $h(t)$ is clearly greater than one, t cannot be a variable symbol or a constant—it must be a complex term of the form $t_0(t_1, \ldots, t_n)$ where $n \in N$, $n \geq 1$ and for all $i \in N$, $0 \leq i \leq n$, t_i is a *HiLog* term with $h(t_i) \leq m$. Thus the condition of line 3 tests true. Now t' must be a *HiLog* ground term of the form $t'_0(t'_1, \ldots, t'_n)$ where, for all $i \in N$, $0 \leq i \leq n$, t'_i is a *HiLog* ground term. If this were not the case, the condition of line 4 would test true and match(t, t') would return FALSE. It follows that the compound statement of lines 7–12 is executed. Now every recursive call to match in the for-statement of lines 8–10 must return TRUE, since otherwise match(t, t') would return FALSE. It follows from the inductive hypothesis that, for every $i \in N$, $0 \leq i \leq n$, there exists a valid variable assignment σ_i over the variables of t_i s.t. $t_i\sigma_i = t'_i$. Now let $\sigma = \bigcup_{0 \leq i \leq n} \sigma_i$. It can be shown that σ is a valid variable assignment as follows: Assume that σ is *not* a valid variable assignment. Then, for some variable v in t and some pair of distinct ground terms g_1 and g_2, σ must contain both (v, g_1) and (v, g_2). Hence, there exist $i \in N$ and $j \in N$, where $0 \leq i < j \leq n$, s.t. the valid variable assignment σ_i contains (v, g_1) and the valid variable assignment σ_j contains (v, g_2), and both t_i and t_j contain v. By the inductive hypothesis $t_i\sigma_i = t'_i$

and, prior to the execution of match(t_i, t_i'), $\nu_{t_i} \subseteq \sigma_i$, so, by Theorem 4, $\nu_{t_i} = \sigma_i$ after the execution of match(t_i, t_i'). Thus, prior to the execution of match(t_j, t_j'), ν, and thus also ν_{t_j}, contains the pair (v, g_1). Now note that, by the inductive hypothesis, ν_{t_j} is a subset of σ_j prior to the execution of match(t_j, t_j'), so σ_j must also include (v, g_1). Since σ_j also contains (v, g_2), it is an invalid variable assignment. This contradicts the inductive hypothesis (which requires σ_j to be valid) and forces the conclusion that σ is indeed a valid variable assignment. Since σ is the union of all the σ_i, and since each σ_i binds all the variables in t_i, and only those variables, it follows that σ binds all the variables in t, and only those variables. Furthermore, for each i, σ_{t_i} (i.e. σ restricted to the variables of t_i) is clearly equal to σ_i, so that $t_i\sigma = t_i\sigma_i = t_i'$ Thus $t\sigma = t_0\sigma(t_1\sigma, \ldots, t_n\sigma) = t_0'(t_1', \ldots, t_n') = t'$. To complete the proof of the inductive step, it suffices to show that, prior to the execution of match(t, t'), $\nu_t \subseteq \sigma$. Assume that ν_t is not a subset of σ. Then ν must contain a pair (v, g_1), where v is a variable in t and g_1 is a ground term, and σ must contain a pair (v, g_2) where g_2 is a ground term distinct from g_1. Therefore, for some $i \in N$, $0 \leq i \leq n$, σ_i must contain (v, g_2). But, prior to the evaluation of match(t_i, t_i'), ν_{t_i} will contain (v, g_1) and, since it follows from the inductive hypothesis that $\nu_{t_i} \subseteq \sigma_i$, σ_i must also contain (v, g_1) and so σ_i must be invalid. This contradicts the inductive hypothesis (which requires σ_i to be valid) and forces the conclusion that, prior to execution of match(t, t'), $\nu_t \subseteq \sigma$.

For the basis of the induction, i.e. $h(t) = 1$, observe that t must be a variable or a constant.

If t is a variable, say v, then $\sigma = \{(v, t')\}$ is a valid variable assignment over the variables of t s.t. $t\sigma = t'$. Since match(t, t') returns TRUE and since the condition of line 14 must test true, there only two possibilities:

1. the conditions of lines 15 and 16 both test true or
2. the condition of line 15 tests false.

In the former case ν_t must clearly be $\{(v, t')\}$, and so $\nu_t \subseteq \sigma$. In the latter case $\nu_t = \emptyset$, and so $\nu_t \subseteq \sigma$. This completes the proof of the basis for the case where t is a variable.

If t is constant then, since match(t, t') returns TRUE, the condition on line 26 must test true and so t' must be a constant equal to t. Then $\sigma = \emptyset$ is a valid variable assignment over the variables of t s.t. $t\sigma = t'$ and, since $\nu_t = \emptyset$, ν_t is clearly a subset of σ. □

Theorems 4 and 5 prove that match is both correct and complete.

3.2.2 The Naive Rule Application Algorithm

Recall that applying a rule to a set of facts entails finding variable assignments under which a given set of facts satisfies the body of the rule, substituting for the variables in the rule head, and then adding the resulting ground terms to the set. To facilitate a more formal discussion of this process, it is necessary to define a *rule transform function* (cf. [21]) which maps a set of facts onto the set of new facts generated by the rule application.

Definition 7 (Rule Transforms) *Let L be a language of HiLog with Herbrand Universe H_L. Let c be a definite HiLog Horn clause $A_0 \vee \neg A_1 \vee \ldots \vee \neg A_n$ in L, where $n \in N$, $n \geq 1$ and $A_0, \ldots, A_n$ are HiLog atomic formulas. Then T_c is a function defined over $\mathcal{P}(H_L)$ and having values in $\mathcal{P}(H_L)$. Specifically, if $I \in \mathcal{P}(H_L)$, then $T_c(I)$ is the set of all α in H_L s.t. there exists a variable assignment ν under which $\alpha = A_0\nu$ and $A_1\nu, \ldots, A_n\nu$ are all elements of I.* □

Now the purpose of the naive rule application algorithm can be stated formally in terms of the rule transform function as follows: given a set I of *HiLog* ground terms and a definite *HiLog* Horn clause c, add to I the ground terms of $T_c(I)$.

Example 1 illustrated the application of a simple rule containing only one subgoal. To see how this process may be extended to deal with rules containing two or more subgoals, consider the following example.

Example 7 Let c be the *HiLog* rule $s(X, Z) :- p(f(X), Y), q(Y, Z)$ and let I be the set of facts $\{p(f(a), b), p(f(a), c), q(b, c), q(b, d)\}$. To apply c to I, it is necessary to find assignments to the variables X, Y and Z which *simultaneously* make both subgoals of the rule true.

Consider the following variable assignments:

$$\nu_1 = \{(X, a), (Y, b)\}$$
$$\nu_2 = \{(X, a), (Y, c)\}$$

$$\mu_1 = \{(Y, b), (Z, c)\}$$
$$\mu_2 = \{(Y, b), (Z, d)\}$$

Note that ν_1 and ν_2 both make the first subgoal true, since they make it identical to facts $p(f(a), b)$ and $p(f(a), c)$ respectively. Similarly, μ_1 and μ_2 make the second subgoal true, since they make it identical to facts $q(b, c)$ and $q(b, d)$ respectively. Now observe that ν_1 is "compatible" with μ_1, in the sense that both assignments bind the same constant value, b, to variable Y. Thus it is possible to combine ν_1 and μ_1 to produce a single variable assignment, $\{(X, a), (Y, b), (Z, c)\}$, which simultaneoulsy makes both subgoals of the rule true. Similarly ν_1 and μ_2 can be combined to produce the variable assignment $\{(X, a), (Y, b), (Z, d)\}$ which also makes both subgoals true. However, ν_2 cannot be combined with either μ_1 or μ_2, since ν_2 binds the value c to Y, while μ_1 and μ_2 both bind the value b to Y.

It is not difficult to see that this process of finding "compatible variable assignments", and then combining them to produce assignments over the variables of both subgoals, is comparable to the process of computing the natural join of two relations. Specifically, ν_1 and ν_2 may be represented by the tuples (a, b) and (a, c), respectively, in a relation r_1 defined over the relation scheme (X, Y). Similarly, μ_1 and μ_2 may be represented by the tuples (b, c) and (b, d), respectively, in a relation r_2 defined over the relation scheme (Y, Z). Then $(r_1 \bowtie r_2)$ is a relation over the scheme (X, Y, Z) and contains the tuples (a, b, c) and (a, b, d). Note that these tuples represent the variable assignments $\{(X, a), (Y, b), (Z, c)\}$ and $\{(X, a), (Y, b), (Z, d)\}$ which were derived intuitively by "combining compatible variable assignments."

The final step of the rule application involves taking each of the variable assignments represented by $(r_1 \bowtie r_2)$, substituting for the variables in the head of the rule and adding the resulting ground terms, i.e. $s(a, c)$ and $s(a, d)$, to I. □

Example 7 suggests that it is possible to apply any rule to a set I of ground terms as follows:

- for each subgoal A_i in the body of the rule, use term-matching (against the elements of I) to compute a relation r_i whose tuples represent variable assignments under which I satisfies A_i;

- compute the natural join of the r_i relations to obtain a relation r_{body} whose tuples represent variable assignments under which I satisfies the *entire rule body*;
- for each tuple in r_{body}, substitute for the variables in the head of the rule and add the resulting ground term to I.

The procedure is similar to that described for Datalog with function symbols in [35].

Readers familiar with the naive evaluation of Datalog, as described in, for example, [34], will have noticed that, while the rule application procedures typically employed by naive evaluation of Datalog delay the addition of newly-generated facts to the sets representing the evolving model until the end of an iteration of naive evaluation, the algorithm described here adds newly-derived facts to the model immediately. This is done intentionally and later helps to illustrate the argument in favour of the GSN evaluation algorithm of Chapter 6.

The remainder of this section describes and validates an algorithm for performing this procedure. The discussion relies on Definitions 8 and 9 below, which formalize the equivalence of variable assignments and relational tuples.

Definition 8 (Tuple of a variable assignment) *Let ν be a variable assignment which binds all the variables in the set $\{v_1, \ldots, v_n\}$ and only those variables. Let u be an element of a relation defined over the relation scheme $(v_1, \ldots, v_n)$ and assume that, for every $i \in N$, $1 \leq i \leq n$, $u[v_i] = \nu(v_i)$. Then the tuple u is said to "denote variable assignment ν" and is written as τ_ν.* □

Definition 9 (Variable assignment of a tuple) *Let $\{v_1, \ldots, v_n\}$ be a set of variable symbols. Let u be an element of a relation over the relation scheme $(v_1, \ldots, v_n)$. Let ν be the variable assignment $\{(v_1, u[v_1]), \ldots, (v_n, u[v_n])\}$, i.e. the set which includes the ordered pair $(v_i, u[v_i])$ for each $i \in N$, $1 \leq i \leq n$, and which includes no other ordered pairs. Then ν is said to be "the variable assignment denoted by u" and is written as ψ_u.* □

The algorithm for rule application is based on the procedure apply (see Figure 3.2), which accepts a definite *HiLog* Horn clause c as an argument and operates in the presence of a

```
1:  void apply(c)
    /* c is a definite HiLog Horn clause; I is a global set of HiLog ground
    terms; ν is the global variable assignment accessed by match. */
2:  {
        /* Let c be the clause A_0 ∨ ¬A_1 ∨ ... ∨ ¬A_n, where n ∈ N,
        n ≥ 1 and A_0, ..., A_n are nonground HiLog atomic formulas. */
3:      for (i = 1; i <= n; i++)
            /* Let the set of distinct variables in A_i be
            {v_{i_1}, ..., v_{i_{m_i}}}. */
4:          create an empty relation r_i with scheme (v_{i_1}, ..., v_{i_{m_i}});

        /* Let I be the set {t_1, ..., t_p}. */
5:      for (j = 1; j <= p; j++)
6:          for (k = 1; k <= n; k++)
7:          {
8:              ν = ∅;
9:              if (match(A_k, t_j))
10:                 r_k = r_k ∪ {τ_ν};
11:         }

12:     create a relation r_body = Π_{v_1,...,v_q}(r_1 ⋈ ... ⋈ r_n) where
13:     {v_1, ..., v_q} is the set of distinct variables in A_0;

        /* Let r_body be the set {u_1, ..., u_w} */
14:     for (h = 1; h <= w; h++)
15:         I = I ∪ {A_0 ψ_{u_h}};
16: }
```

Figure 3.2: Procedure for naive rule application

global set I of *HiLog* ground terms and the global variable assignment ν. The procedure adds the elements of $T_c(I)$ to the global set I.

Theorems 6 and 7 below prove that the apply function is both complete and correct. The first theorem defines, for a given definite *HiLog* Horn clause c and a given set I of *HiLog* ground terms, a relational algebra expression $\Gamma_{c,I}$, and then proves that it is equal to $T_c(\mathrm{I})$. The second theorem proves that, if apply is invoked with c as an argument and in the presence of a global set I of *HiLog* ground terms, then the set of terms added to I is precisely equal to $\Gamma_{c,I}$.

Theorem 6 *Let L be a language of HiLog with Herbrand Universe H_L. Let $I \in \mathcal{P}(H_L)$. Let c be the definite HiLog Horn clause $A_0 \vee \neg A_1 \vee \ldots \vee \neg A_n$, where $n \in N$, $n \geq 1$ and $A_0, \ldots, A_n$ are nonground HiLog terms. Assume that the set of distinct variable symbols in A_0 is $\{v_1, \ldots, v_q\}$ and that, for each $i \in N$, $1 \leq i \leq n$, the set of distinct variable symbols in A_i is $\{v_{i_1}, \ldots, v_{i_{m_i}}\}$. Let $\Gamma_{c,I} = \{A_0\mu \mid \tau_\mu \in \pi_{v_1,\ldots,v_q}(r_1 \bowtie \ldots \bowtie r_n)\}$, where, for all $i \in N$, $1 \leq i \leq n$, r_i is defined as follows: r_i is a relation over the relation scheme $(v_{i_1}, \ldots, v_{i_{m_i}})$, each attribute of which has domain H_L; specifically, $r_i = \{u_i \in {H_L}^{m_i} \mid A_i\psi_{u_i} \in I\}$. Then $\Gamma_{c,I} = T_c(I)$.*

Proof: ($T_c(I) \subseteq \Gamma_{c,I}$): Let the *HiLog* ground term t be an element of $T_c(I)$. It follows from the definition of T_c that there exists a variable assignment ν under which $t = A_0\nu$ and $A_1\nu, \ldots, A_n\nu$ are all elements of I. Observe that, for any $i \in N$, $1 \leq i \leq n$, $A_i\nu = A_i\nu_{A_i}$, so $A_i\nu_{A_i} \in I$. It follows from the definition of r_i that r_i contains a tuple u_i s.t. $\psi_{u_i} = \nu_{A_i}$. Observe that, for any variable symbol v in A_i, $u_i[v] = \psi_{u_i}(v) = \nu_{A_i}(v) = \nu(v)$. Specifically, let j and k be any natural numbers s.t. $1 \leq j < k \leq n$ and let v be any variable symbol which occurs in both A_j and A_k. Then $u_j[v] = u_k[v]$, since both are equal to $\nu(v)$. Thus $u_1, \ldots, u_n$ satisfy the join conditions and yield, in $r_1 \bowtie \ldots \bowtie r_n$, a tuple, say s, s.t. $s[v] = \nu(v)$ for any variable symbol v in c. So $\pi_{v_1,\ldots,v_q}(r_1 \bowtie \ldots \bowtie r_n)$ includes a tuple, say p, s.t., for any variable symbol v in A_0, $p[v] = s[v] = \nu(v)$. It follows from the definition of $\Gamma_{c,I}$ that $\Gamma_{c,I}$ includes a ground term $A_0\mu$ s.t. $\tau_\mu = p$. Note that, for any variable symbol v in A_0, $\mu(v) = \tau_\mu[v] = p[v] = \nu(v)$. Thus $A_0\mu = A_0\nu$ and, since $A_0\nu = t$, t is an element of $\Gamma_{c,I}$.

($\Gamma_{c,I} \subseteq T_c(I)$): Let the *HiLog* ground term t be an element of $\Gamma_{c,I}$. It follows from the definition of $T_c(I)$ that, in order to prove that $t \in T_c(I)$, it suffices to prove that there exists a variable assignment ν s.t. $A_0\nu = t$ and, for each $i \in N$, $1 \leq i \leq n$, $A_i\nu \in I$. From the definition of $\Gamma_{c,I}$, there exists a variable assignment μ s.t. $A_0\mu = t$ and $\tau_\mu \in \pi_{v_1,\ldots,v_q}(r_1 \bowtie \ldots \bowtie r_n)$. Thus $r_1 \bowtie \ldots \bowtie r_n$ must contain a tuple, say s, s.t., for every variable symbol v in A_0, $s[v] = \tau_\mu[v] = \mu(v)$. Now let $\nu = \psi_s$. Then, for every variable symbol v in c, $\nu(v) = \psi_s(v) = s[v]$. In particular, for every variable symbol v in A_0, $\nu(v) = s[v] = \mu(v)$. Thus $A_0\nu = A_0\mu = t$. All that remains is to prove that $A_1\nu, \ldots, A_n\nu$ are all elements of I. Observe that, for each $i \in N$, $1 \leq i \leq n$, r_i must contain a tuple u_i s.t., for every variable symbol v in A_i, $u_i[v] = s[v]$, since otherwise s would not be in $r_1 \bowtie \ldots \bowtie r_n$. Now it follows from the definition of r_i that $A_i\psi_{u_i} \in I$. For every variable symbol v in A_i, $\psi_{u_i}(v) = u_i[v] = s[v] = \nu(v)$. Therefore $A_i\psi_{u_i} = A_i\nu$ and so $A_i\nu \in I$ and the proof is complete. □

Theorem 7 *The function* apply *is correct and complete in that, for any definite HiLog Horn clause c denoting a HiLog rule, if* apply *is invoked with argument c and in the presence of the global set I of HiLog ground terms, then* apply *will add to I all the elements of $T_c(I)$ and only those elements.*

Proof: Let c be a definite *HiLog* Horn clause in the *HiLog* language L, where L has Herbrand Universe H_L. Specifically, let $c = A_0 \vee \neg A_1 \vee \ldots \vee \neg A_n$, where $n \in N$, $n \geq 1$ and $A_0, \ldots, A_n$ are nonground *HiLog* terms. Let the set of distinct variable symbols in A_0 be $\{v_1, \ldots, v_q\}$ and, for each $i \in N$, $1 \leq i \leq n$, let the set of distinct variable symbols in A_i be $\{v_{i_1}, \ldots, v_{i_{m_i}}\}$. Then, by Theorem 6, $T_c(I) = \{A_0\mu \mid \tau_\mu \in \pi_{v_1,\ldots,v_q}(r'_1 \bowtie \ldots \bowtie r'_n)\}$, where, for all $i \in N$, $1 \leq i \leq n$, r'_i is a relation over the relation scheme $(v_{i_1}, \ldots, v_{i_{m_i}})$, each attribute of which has domain H_L. Specifically, $r'_i = \{u_i \in {H_L}^{m_i} \mid A_i\psi_{u_i} \in I\}$.

Now observe that the for-loop of lines 3 and 4 creates r_i as a relation over the same scheme over which r'_i is defined. Also, the compound statement of lines 7–11 is executed for each t_j in I and each A_k in the body of c and, since match is complete and correct by Theorems 4 and 5, line 10 adds τ_ν to to r_k iff $A_k\nu = t_j$. Since r'_k can be rewritten as $\{\tau_\nu \in {H_L}^{m_i} \mid A_k\nu \in I\}$, it follows that, after execution of the nested for-loop of lines 5–11, $r_i = r'_i$ for all $i \in N$, $1 \leq i \leq n$.

```
1:  void least_model(F, P)
    /* F is a finite set of HiLog ground terms; P is a finite set of
    definite HiLog Horn clauses; I is a global set of HiLog ground terms. */
2:  {
3:      I = ∅;

        /* Let F be the set {t_1, ..., t_n}. */
4:      for (i = 1; i <= n; i++)
5:          I = I ∪ {t_i};

6:      while (I has changed)
            /* Let P be the set {c_1, ..., c_m}. */
7:          for (j = 0; j <= m; j++)
8:              apply(c_j);
9:  }
```

Figure 3.3: Procedure for naive evaluation

Therefore the r_{body} relation computed on line 12 is equal to $\pi_{v_1,\ldots,v_q}(r'_1 \bowtie \ldots \bowtie r'_n)$ and, since $T_c(I)$ can be rewritten as $\{A_0\psi_u \mid u \in \pi_{v_1,\ldots,v_q}(r'_1 \bowtie \ldots \bowtie r'_n)\}$, the ground terms added to I on line 15 are precisely the ground terms of $T_c(I)$. □

3.2.3 The Naive Evaluation Algorithm

Naive evaluation is a simple scheme for computing the least Herbrand model of a set of *HiLog* facts and a *HiLog* program comprising only definite Horn clause rules. The algorithm is based on the procedure least_model (see Figure 3.3) which accepts as arguments a set F of *HiLog* ground terms and a set P of definite *HiLog* Horn clauses. The procedure operates in the presence of the global set I of *HiLog* ground terms and, if the least model M of F and P is finite, it terminates with $I = M$. Note, once again, that the evaluation doesn't maintain a separation of the newly-generated facts from the evolving model until the end of the iteration, as is done in the procedure for naive evaluation of Datalog described in [34].

In order to prove that least_model is complete and correct, it is necessary to establish two important properties of the T_c function. Theorem 8 proves that it is not possible to apply a rule of a program to a set of facts which is a subset of the least model of the program

and obtain a fact which is not in that least model. Theorem 9 proves that, if no rule of a program P can be applied to a set of *HiLog* ground terms I to generate new ground terms, then I must satisfy all the rules of P.

Theorem 8 *Let L be a language of HiLog with Herbrand Universe H_L. Let $F \in \mathcal{P}(H_L)$. Let P be a program in L, defined in terms of a set of rules, each of which is, in turn, defined in terms of a definite HiLog Horn clause. Let M be the least Herbrand model of F and P. Let $I \in \mathcal{P}(H_L)$ and assume that $F \subseteq I$ and $I \subseteq M$. Then, for any r in P, $T_r(I) \subseteq M$.*

Proof: Let r be the definite *HiLog* Horn clause $A_0 \vee \neg A_1 \vee \ldots \vee \neg A_n$, where $n \in N$, $n \geq 1$ and $A_0, \ldots, A_n$ are *HiLog* atomic formulas. Let $\alpha \in H_L$ and assume that $\alpha \in T_r(I)$. Then, by the definition of T_r there exists a variable assignment ν s.t. $\alpha = A_0\nu$ and $A_1\nu, \ldots, A_n\nu$ are all elements of I. Since $I \subseteq M$, and since M is the intersection of all Herbrand models of F and P, it follows that $A_1\nu, \ldots, A_n\nu$ are all elements of every Herbrand model of F and P. But each such Herbrand model of F and P is required to satisfy r under all variable assignments, including ν. It follows that $A_0\nu$ must be an element of every Herbrand model of F and P and so $A_0\nu$ must be an element of M. This completes the proof, since $\alpha = A_0\nu$. □

Theorem 9 *Let L be a HiLog language with Herbrand universe H_L. Let $F \in \mathcal{P}(H_L)$. Let P be a program in L, defined in terms of a set of rules, each of which is, in turn, defined in terms of a definite HiLog Horn clause. Let $I \in \mathcal{P}(H_L)$ and assume that $F \subseteq I$. Assume, furthermore, that for every r in P, $T_r(I) \subseteq I$. Then I is a model of F and P.*

Proof: Assume that I is not a model of P and F. Since, by definition, $F \subseteq I$, I must fail to be a model because it does not satisfy some clause in P. Specifically, P must include a clause r, where $r = A_0 \vee \neg A_1 \vee \ldots \vee \neg A_n$, s.t. there exists a variable assignment ν under which $A_1\nu, \ldots, A_n\nu$ are all elements of I and $A_0\nu$ is not an element of I. But it follows from the definition of T_r that $T_r(I)$ must include $A_0\nu$, so $T_r(I) \not\subseteq I$. This contradicts the assumption that, for every r in P, $T_r(I) \subseteq I$. The contradiction forces the conclusion that I is indeed a model of F and P. □

The correctness and completeness of the naive evaluation algorithm now follow easily from Theorems 8 and 9.

Theorem 10 *Given any finite set F of HiLog ground terms and any finite set P of definite HiLog Horn clauses, if the least model M of F and P is finite, then* least_model(F, P) *duly terminates with $I = M$.*

Proof: It is straightforward to show that, throughout the execution of the while-loop of lines 6–8, it remains the case that $F \subseteq I$ and $I \subseteq M$. The proof is an induction on the number of calls n to procedure apply executed on line 8: For the basis, i.e. $n = 0$, observe that, just prior to the execution of the while-loop, $I = F$. Thus $F \subseteq I$ and, furthermore, since $F \subseteq M$, $I \subseteq M$. For the inductive step, assume that the assertion is true for $n = i$, i.e. $F \subseteq I \subseteq M$ after i calls to procedure apply. Now assume that the next call to apply, say apply(c), where $c \in P$, adds a ground term t to I. It follows from Theorem 7 that $t \in T_c(I)$. Observe that, by the inductive hypothesis, $I \subseteq M$ and so, by Theorem 8, t must be an element of M. Therefore, after $(i + 1)$ calls to apply it remains the case that $F \subseteq I \subseteq M$ and the inductive step is proved.

Now note that, since I remains a subset of M and M is finite, I cannot increase in size indefinitely—eventually the condition of line 6 will test false and the algorithm will terminate. When this is the case $T_c(I)$ must be a subset of I for every $c \in P$. If not, at least one call to apply in the last iteration of the while-loop would have added a new term to I (by Theorem 7) and the condition of line 6 would have tested true. It follows from Theorem 9 that I is a model of F and P and, since $I \subseteq M$, I is the *least* model of F and P. □

3.3 The proto System

The proto system is a simple deductive database system that enables a user to define, reason about and query data using the restricted *HiLog* language described in Section 2.5. It is based on the naive evaluation algorithm described in this chapter. The system is implemented on a relational database platform (the *Informix* RDBMS) in that:

- it uses database tables maintained by the RDBMS to store sets of *HiLog* ground terms and
- it uses the SQL query language supported by the RDBMS to perform the relational algebra operations required by the rule application procedure.

3.3.1 System Organization

The main components of the system are as follows:

Lexical Analyser Written in C, it converts a stream of input characters into a stream of tokens.

Parser Based on a recursive-descent algorithm and written in C, it analyses input and constructs an internal data structure to represent a valid *HiLog* program.

Pre-evaluation component This is also written in C and prepares the internal data structure prior to the execution of the evaluation component. Specifically, it creates the SQL queries required by the rule application procedures.

Evaluation component This is written in "C with embedded SQL" and uses the interface provided by the RDBMS (Informix) and the SQL queries generated by the pre-evaluation component to implement the naive evaluation algorithm. It is based on apply (Figure 3.2) and least_model (Figure 3.3).

3.3.2 Database Usage

The system uses a single-column table to store the evolving Herbrand model and, during application of a rule, it uses a table, whose columns correspond to subgoal variables, for each of the "r_i relations" described in Figure 3.2. In each case, string values are used to represent the *HiLog* ground terms.

Chapter 4

Seminaive Evaluation

This chapter describes *seminaive evaluation*, an algorithm for the bottom-up evaluation of *HiLog* which can be substantially more efficient than the naive evaluation algorithm described in the previous chapter. Section 4.1 presents a largely intuitive overview of this second approach to *HiLog* evaluation, while Section 4.2 details algorithms for seminaive rule application and for computing the least Herbrand model of a given set of facts and rules using seminaive evaluation. Section 4.3 describes the semi system, a modified version of the proto system (Section 3.3) based on seminaive evaluation. Finally, Section 4.4 compares the performance of seminaive evaluation with that of naive evaluation, employing both a theoretical analysis and data generated by the proto and semi systems.

4.1 An Overview of Seminaive Evaluation

This section presents an informal introduction to the seminaive evaluation of logic programs, discussed in the context of Datalog evaluation in [6, 5, 9]. Example 8 below uses a straightforward program to demonstrate the shortcomings of the naive evaluation algorithm described in the previous chapter. The remainder of the section states the objectives of seminaive evaluation in terms of *derivations* and the *non-repetition property* and uses the program of Example 8 to illustrate the principles and techniques of seminaive evaluation.

Example 8 Let F be a set of facts

$$\{e(a,c), e(b,c), e(c,d), e(d,e), e(a,e)\}$$

denoting the edge set of a directed graph and let P be a pair of rules

$$\begin{aligned} r_1\text{:}\quad & p(X,Y) \quad :- \quad e(X,Y) \\ r_2\text{:}\quad & p(X,Z) \quad :- \quad e(X,Y), p(Y,Z) \end{aligned}$$

defining paths in the graph. Now consider the use of naive evaluation to compute the least Herbrand model of F and P.

The evaluator begins by approximating the model as the set F. Since every fact in this set matches the subgoal of the first rule, the application of the first rule on the first iteration of naive evaluation computes the set of facts

$$\{p(a,c), p(b,c), p(c,d), p(d,e), p(a,e)\}$$

and adds these facts to the model. Then, when the second rule is applied, three further facts are computed and added to the model: $e(a,c)$ and $p(c,d)$ yield $p(a,d)$; $e(b,c)$ and $p(c,d)$ yield $p(b,d)$; $e(c,d)$ and $p(d,e)$ yield $p(c,e)$.

Now consider the application of the first rule at the beginning of the *second* iteration of the naive evaluation. Since the rule application procedure always uses *all* the facts in the model to compute derived facts, the evaluator *again* uses the set of facts

$$\{e(a,c), e(b,c), e(c,d), e(d,e), e(a,e)\}$$

to derive the set of facts

$$\{p(a,c), p(b,c), p(c,d), p(d,e), p(a,e)\}$$

even though these latter facts were computed in exactly the same way on the first iteration. Consider too the application of the second rule on the second iteration of the evaluation. The evaluator uses $e(a,c)$ and $p(c,e)$ to derive $p(a,e)$, and uses $e(b,c)$ and $p(c,e)$ to derive $p(b,e)$. Since $p(c,e)$ was not in the model when the second rule was first applied, these derivations are clearly new. (It is worth noting that the derivation of $p(a,e)$ is regarded as "new" even though $p(a,e)$ is not a new fact, because prior derivations of the fact involved

application of the first rule, rather than the second.) However, the evaluator also repeats the derivations which it performed on the first iteraton, i.e. the derivations of $p(a, d)$, $p(b, d)$ and $p(c, e)$, because it has no way of distinguishing new combinations of facts from old. Similarly, all the derivations performed on the second iteration are repeated on the third iteration, even though the third iteration does not compute any new facts. □

Repetition of derivations, as illustrated in the above example, significantly compromises the efficiency of naive evaluation. The seminaive evaluation algorithm described in this chapter is said to exhibit the *non-repetition property* because it successfully avoids such repeated derivations. Formal definitions of the notions of "derivation," "performing a derivation" and "the non-repetition property" follow. See also [30].

Definition 10 *Let I be a set of HiLog ground terms and let c be the definite HiLog Horn clause $A_0 \vee \neg A_1 \vee \ldots \vee \neg A_n$, where $n \in N$, $n \geq 1$ and $A_0, \ldots, A_n$ are nonground HiLog terms. Now let $t \in T_c(I)$, so that there exists a variable assignment ν under which $A_0\nu = t$ and $A_1\nu, \ldots, A_n\nu$ are all elements of I. Then the ordered pair (c, ν) denotes a derivation of t under c and I.* □

Assume that apply, the function for naive rule application detailed in Figure 3.2, is invoked with the definite *HiLog* Horn clause c as an argument. Now observe that each tuple of the join computed on line 12 of the function, i.e. each tuple of $(r_1 \bowtie \ldots \bowtie r_n)$, represents an assignment over the variables of c which makes the body of c true. Thus each such tuple corresponds to a derivation of some fact in the evolving Herbrand model. Note that, since the algorithm performs work each time it computes such a tuple, the process of computing the tuple is regarded as "performing a derivation," regardless of whether or not the tuple yields any new facts for addition to the evolving model.

Definition 11 below presents a general definition, applicable to all the evaluation schemes discussed in this work, of "performing a derivation."

Definition 11 *Consider any function which accepts, as an argument, a definite HiLog Horn clause c and which applies c to some set I of facts in order to generate new facts. Each time the function computes a tuple which represents a variable assignment ν under which I satisfies the body of c, the function is said to "perform the derivation (c, ν)."* □

Definition 12 *Consider any algorithm for computing the least Herbrand model of a set of HiLog facts and a set of definite HiLog Horn clause rules. If the algorithm does not perform any derivation more than once during the evaluation, it is said to exhibit the non-repetition property.* □

Example 8 pointed out that the naive evaluation algorithm repeats derivations because it is unable to distinguish new combinations of facts from old ones. This suggests that any evaluation algorithm which seeks to avoid repeated derivations will have to somehow partition the evolving model into a set of "new" facts and a set of "old" facts. In particular, when a rule is applied to the model, the rule-application procedure must be able to distinguish between facts which have been "seen" by the rule, in that they were present in the model when the rule was last applied, and those which have not yet been seen by the rule, in that they were only added to the model after the rule was last applied. Seminaive evaluation achieves this by using *two* sets to store the facts of the evolving model: immediately before the program rules are applied during an iteration of seminaive evaluation, I^{old} contains those facts which have already been seen by *every* rule of the program, while I^{Δ} contains those facts which have not yet been seen by *any* rule of the program.

During an iteration, each rule of the program is applied to the model by a seminaive rule application procedure which generates all the new facts which can be derived from $(I^{old} \cup I^{\Delta})$ and stores them in a third set, I^{new}. By taking advantage of the partitioning of the model into sets of new and old facts, this procedure examines only *new* combinations of facts and thus avoids repeating any prior derivations.

The I^{new} set is intended to accumulate all those facts which can be derived by applying the program rules to $(I^{old} \cup I^{\Delta})$, but to exclude any facts which are already in $(I^{old} \cup I^{\Delta})$. It is important to note that the non-repetition property only ensures that *derivations* are not repeated—because each fact may have several distinct derivations, it does not guarantee that a given fact won't be computed more than once during the evaluation. This was illustrated in Example 8, where the fact $p(a, e)$ was derived both by applying the first rule to the single fact $e(a, e)$ and by applying the second rule to the pair of facts $e(a, c)$ and $p(a, c)$. Thus, in order to ensure that I^{new} and $(I^{old} \cup I^{\Delta})$ are indeed disjoint at the end of an iteration, the algorithm subtracts $(I^{old} \cup I^{\Delta})$ from I^{new} after applying all the program

rules.

It is not difficult to see that, at this stage, each fact in $(I^{old} \cup I^{\Delta})$ has been seen by *every* program rule, while each fact in I^{new} is unseen by *any* program rule. Thus the algorithm assimilates the facts of I^{Δ} into I^{old} and, at the beginning of the next iteration, sets I^{Δ} equal to I^{new} and I^{new} equal to the empty set.

The algorithm terminates when an examination of I^{new} between iterations finds the set to be empty. Later sections of this chapter prove that, when this condition is met, the value of I^{old} is precisely equal to the required model.

Example 9 Consider again the program of Example 8: F is a set of facts

$$\{e(a,c), e(b,c), e(c,d), e(d,e), e(a,e)\}$$

denoting the edges of a directed graph; P is a pair of rules

$$\begin{aligned} r_1\text{:}\quad & p(X,Y) \quad :\!- \quad e(X,Y) \\ r_2\text{:}\quad & p(X,Z) \quad :\!- \quad e(X,Y), p(Y,Z) \end{aligned}$$

defining paths in the graph. Computation of the least Herbrand model of F and P by means of *seminaive evaluation* is described below.

When the rules are applied on the first iteration, I^{old}, the set of "old" facts, is equal to the empty set, while I^{Δ}, the set of "new" facts, is equal to the set F of given facts, i.e.

$$\{e(a,c), e(b,c), e(c,d), e(d,e), e(a,e)\}.$$

Since all these facts match the subgoal of the first rule, the application of the first rule yields the set of facts

$$\{p(a,c), p(b,c), p(c,d), p(d,e), p(a,e)\}$$

which are stored in the I^{new} set. However, since no facts capable of matching the second rule's second subgoal are present in either I^{old} or I^{Δ}, application of the second rule does not produce any new facts.

At the end of the first iteration, the facts of I^{Δ} are assimilated into I^{old} and, at the beginning of the second iteration, the facts of I^{new} are placed in I^{Δ}. Clearly, when the rules are applied on the second iteration, I^{old} contains facts representing the edges of the

graph, while I^{Δ} contains facts representing the paths of unit length in the graph. Now consider the application of the first rule to the database: the facts in the I^{old} relation have all been "seen" by the rule, in that they were present when the rule was applied on the first iteration; thus, in order to avoid repeating derivations, the rule application procedure does *not* consider the facts of the I^{old} relation - it examines only the facts of the I^{Δ} relation and, since none of these facts match the rule's subgoal, it does not generate any new facts. The application of the second rule, however, uses the "edge facts" of the I^{old} relation and the "unit length path facts" of the I^{Δ} relation to produce facts which denote paths of length two in the graph. In particular: $e(a, c)$ and $p(c, d)$ yield $p(a, d)$; $e(b, c)$ and $p(c, d)$ yield $p(b, d)$; $e(c, d)$ and $p(d, e)$ yield $p(c, e)$. It is worth noting that, although each derivation involved a fact in the I^{old} relation, none was a repeated derivation, since each involved a combination of facts which was necessarily "new," in that it included at least one fact drawn from the I^{Δ} relation.

Once again, the facts of I^{Δ} are transferred to I^{old} at the end of the second iteration and the facts of I^{new} are transferred to I^{Δ} at the beginning of the third iteration. Thus, when the rules are applied to the database on the third iteration, I^{old} contains facts denoting graph edges and paths of unit length in the graph, while I^{Δ} contains facts denoting paths of length two in the graph. As on the second iteration, application of the first rule examines only I^{Δ} and, finding no facts which match the rule's subgoal, produces no new facts. Now consider the application of the second rule: were the rule-application procedure to use those facts in I^{old} which represent unit length paths, it would clearly repeat the derivations performed on the second iteration; accordingly, the procedure uses only those facts of I^{old} which denote graph edges, in conjunction with the facts of I^{Δ}, to derive new facts denoting paths of length three in the graph. In particular: $e(a, c)$ and $p(c, e)$ yield $p(a, e)$; $e(b, c)$ and $p(c, e)$ yield $p(b, e)$. Since each combination of facts examined by the procedure includes a "new" fact drawn from the I^{Δ} relation, the procedure again avoids repeating any derivations. Note, however, that the non-repetition property does *not* prevent the evaluator from deriving $p(a, e)$ a second time, because the fact has two distinct derivations—it can be attributed to a path of length one or to a path of length three. Nevertheless, the subtraction of $(I^{old} \cup I^{\Delta})$ from I^{new} at the end of the third iteration removes the fact from I^{new} so that it does not appear as a "new fact" on the

fourth iteration.

At the end of the third iteration, the facts of I^{Δ} are assimilated into I^{old} and, at the beginning of the fourth iteration, the facts of I^{new} are transferred to I^{Δ}. Clearly, when the rules are applied on the fourth iteration, I^{old} contains facts denoting graph edges and paths of lengths one and two in the graph, while I^{Δ} contains only the fact $p(b, e)$. As on the second and third iterations, the application of the first rule fails to produce any new facts. When applying the second rule, the rule-application procedure again avoids using any facts in I^{old}, with the exception of those which denote graph edges. Since none of these facts have b as the second argument, none can combine with the $p(b, e)$ fact in I^{Δ}, and so application of the second rule also fails to produce any new facts.

At the end of the fourth iteration, the $p(b, e)$ fact of I^{Δ} is placed in the I^{old} relation and, since I^{new} remains empty at the end of this iteration, evaluation then terminates without any further alterations being made to the relations. It is easily verified that the final value of I^{old} is the set

$$\{p(a, c), p(b, c), p(c, d), p(d, e), p(a, e), p(a, d), p(b, d), p(c, e), p(b, e)\}$$

and that this set is indeed the least model of F and P. □

4.2 Algorithms

4.2.1 The Seminaive Rule Application Algorithm

On any iteration of seminaive evaluation, the seminaive rule application algorithm is invoked for each rule of the program to compute all those facts which may be derived by applying the rule to the current set of database facts, and to do so without repeating any derivation performed on a prior iteration of the evaluation. More specifically, if the database is partitioned into a set I^{old} of facts which have been seen by the rule, and a set I^{Δ} of facts which have not yet been seen by the rule, then the algorithm is required to compute those facts which can be derived by applying the rule to $(I^{old} \cup I^{\Delta})$, without performing any derivation which is based solely on facts in I^{old}.

As for naive rule application, seminaive rule application entails three steps:

- for each subgoal of the rule, perform term-matching against the elements of ($I^{old} \cup I^{\Delta}$); this is to compute a set of tuples representing variable assignments under which ($I^{old} \cup I^{\Delta}$) satisfies the subgoal;
- join the tuples of these sets so as to obtain tuples which represent variable assignments under which ($I^{old} \cup I^{\Delta}$) makes the entire rule body true;
- for each such variable assignment, substitute for the variables of the rule head to derive a fact.

However, the need to avoid performing derivations based exclusively on old facts necessitates partitioning the set of tuples computed for each subgoal into a set of "old tuples" (those derived from old facts) and a set of "new tuples" (those derived from new facts). So, for each subgoal A_i of the rule, the algorithm uses term-matching to compute *two* relations over a common relation scheme whose attributes correspond to the distinct variables in A_i:

- r_i^{old} is a set of tuples representing variable assignments under which I^{old} satisfies A_i;
- r_i^{Δ} is a set of tuples representing variable assignments under which I^{Δ} satisfies A_i.

It is now necessary to join tuples in a manner which considers every combination of tuples comprising exactly one tuple from each subgoal, except those combinations which comprise *only* old tuples. The discussion is reminiscent of the "differentials of relational algebra expressions" described in [6]. Formally, for a rule comprising n subgoals, it is necessary to compute the union of all expressions of the form $q_1 \bowtie \ldots \bowtie q_n$, where, for each $i \in N$, $1 \leq i \leq n$, q_i is either r_i^{old} or r_i^{Δ}, with the exception of the expression in which q_i is r_i^{old} for *all* $i \in N$, $1 \leq i \leq n$. Clearly, there are $2^n - 1$ such expressions, but, fortunately, it turns out that it is possible to exploit the distributivity of the $\bowtie$ and $\cup$ operators to simplify the computation so that it entails taking the union of only n join expressions. In particular, given any $m \in N$, $1 \leq m \leq n$, the union of all 2^{n-m} expressions of the form

$$r_1^{old} \bowtie \ldots \bowtie r_{m-1}^{old} \bowtie r_m^{\Delta} \bowtie q_{m+1} \bowtie \ldots \bowtie q_n$$

can be replaced by the single expression

$$r_1^{old} \bowtie \ldots \bowtie r_{m-1}^{old} \bowtie r_m^{\Delta} \bowtie (r_{m+1}^{old} \cup r_{m+1}^{\Delta}) \bowtie \ldots \bowtie (r_n^{old} \cup r_n^{\Delta}).$$

Thus it follows that it is sufficient to compute

$$\bigcup_{i=1}^{n}(r_1^{old} \bowtie \ldots \bowtie r_{i-1}^{old} \bowtie r_i^{\Delta} \bowtie r_{i+1}^{full} \bowtie \ldots \bowtie r_n^{full})$$

where, for each $i \in N$, $1 \leq i \leq n$,

$$r_i^{full} = r_i^{old} \cup r_i^{\Delta}.$$

Example 10 Assume that it is necessary to apply a rule with *three subgoals* to a database partitioned into a set of old facts, I^{old}, and a set of new facts, I^{Δ}. Matching each of the three subgoals against the facts of I^{old} yields three sets of "old tuples": r_1^{old}, r_2^{old} and r_3^{old}. Similarly, matching each of the three subgoals against the facts of I^{Δ} yields three sets of "new tuples": r_1^{Δ}, r_2^{Δ} and r_3^{Δ}. Also, let $r_1^{full} = r_1^{old} \cup r_1^{\Delta}$, let $r_2^{full} = r_2^{old} \cup r_2^{\Delta}$ and let $r_3^{full} = r_3^{old} \cup r_3^{\Delta}$.

To find all tuples denoting variable assignments under which $(I^{old} \cup I^{\Delta})$ satisfies all three subgoals simultaneously, while avoiding the computation of tuples based exclusively on old facts, it is necessary and sufficient to compute the union of the following seven expressions:

$$r_1^{\Delta} \bowtie r_2^{\Delta} \bowtie r_3^{\Delta} \tag{4.1}$$

$$r_1^{\Delta} \bowtie r_2^{\Delta} \bowtie r_3^{old} \tag{4.2}$$

$$r_1^{\Delta} \bowtie r_2^{old} \bowtie r_3^{\Delta} \tag{4.3}$$

$$r_1^{\Delta} \bowtie r_2^{old} \bowtie r_3^{old} \tag{4.4}$$

$$r_1^{old} \bowtie r_2^{\Delta} \bowtie r_3^{\Delta} \tag{4.5}$$

$$r_1^{old} \bowtie r_2^{\Delta} \bowtie r_3^{old} \tag{4.6}$$

$$r_1^{old} \bowtie r_2^{old} \bowtie r_3^{\Delta} \tag{4.7}$$

But

$$\begin{aligned}
&(4.1) \cup (4.2) = \\
&\quad (r_1^{\Delta} \bowtie r_2^{\Delta} \bowtie r_3^{\Delta}) \cup (r_1^{\Delta} \bowtie r_2^{\Delta} \bowtie r_3^{old}) \\
&= \quad (r_1^{\Delta} \bowtie r_2^{\Delta}) \bowtie (r_3^{old} \cup r_3^{\Delta}) \\
&= \quad r_1^{\Delta} \bowtie r_2^{\Delta} \bowtie r_3^{full} \qquad (4.8)
\end{aligned}$$

Similarly

$$
\begin{aligned}
&(4.3) \cup (4.4) = \\
&\quad (r_1^{\Delta} \bowtie r_2^{old} \bowtie r_3^{\Delta}) \cup (r_1^{\Delta} \bowtie r_2^{old} \bowtie r_3^{old}) \\
&= (r_1^{\Delta} \bowtie r_2^{old}) \bowtie (r_3^{old} \cup r_3^{\Delta}) \\
&= r_1^{\Delta} \bowtie r_2^{old} \bowtie r_3^{full} \qquad (4.9)
\end{aligned}
$$

Then

$$
\begin{aligned}
&(4.8) \cup (4.9) = \\
&\quad (r_1^{\Delta} \bowtie r_2^{\Delta} \bowtie r_3^{full}) \cup (r_1^{\Delta} \bowtie r_2^{old} \bowtie r_3^{full}) \\
&= (r_1^{\Delta} \bowtie r_3^{full}) \bowtie (r_2^{old} \cup r_2^{\Delta}) \\
&= r_1^{\Delta} \bowtie r_2^{full} \bowtie r_3^{full} \qquad (4.10)
\end{aligned}
$$

Also

$$
\begin{aligned}
&(4.5) \cup (4.6) = \\
&\quad (r_1^{old} \bowtie r_2^{\Delta} \bowtie r_3^{\Delta}) \cup (r_1^{old} \bowtie r_2^{\Delta} \bowtie r_3^{old}) \\
&= (r_1^{old} \bowtie r_2^{\Delta}) \bowtie (r_3^{old} \cup r_3^{\Delta}) \\
&= r_1^{old} \bowtie r_2^{\Delta} \bowtie r_3^{full} \qquad (4.11)
\end{aligned}
$$

So the union of expressions 4.1 to 4.7 may be expressed as $(4.10) \cup (4.11) \cup (4.7)$ which is equal to $(r_1^{\Delta} \bowtie r_2^{full} \bowtie r_3^{full}) \cup (r_1^{old} \bowtie r_2^{\Delta} \bowtie r_3^{full}) \cup (r_1^{old} \bowtie r_2^{old} \bowtie r_3^{\Delta})$. □

The remainder of this section provides a more formal description of the seminaive rule application algorithm. It also states and proves a constraint on the set of new facts which the algorithm can generate, so as to provide a basis for proofs of the completeness and correctness of seminaive evaluation in later sections of this chapter.

Figure 4.1 presents an algorithm for seminaive rule application in the form of the pseudocode for a function semi_apply. The function accepts a definite *HiLog* Horn clause c as an argument and operates in the presence of three global sets of *HiLog* ground terms, I^{old}, I^{Δ} and I^{new}, and in the presence of the global variable assignment ν. It does not return a value, but simply adds to I^{new} the facts derived by applying c to $(I^{old} \cup I^{\Delta})$.

```
1:  semi_apply(c)
/* c is a definite Hilog Horn clause; I^old, I^Δ and I^new are global sets
of Hilog ground terms; ν is the global variable assignment accessed
by match. */
2:  {
        /* Let c be the clause A_0 ∨ ¬A_1 ∨ ... ∨ ¬A_n, where n ∈ N,
        n ≥ 1 and A_0, ..., A_n are nonground Hilog atomic formulas. */
3:      for (i = 1; i <= n; i++)
            /* Let the set of distinct variable symbols in A_i be
            {v_{i_1}, ..., v_{i_{m_i}}}. */
4:          create empty relations r_i^old, r_i^Δ and r_i^full over the relation
5:          scheme (v_{i_1}, ..., v_{i_{m_i}});

        /* Let I^old be the set {t_1, ..., t_f}. */
6:      for (j = 1; j <= f; j++)
7:          for (k = 1; k <= n; k++)
8:          {
9:              ν = ∅;
10:             if (match(A_k, t_j))
11:             {
12:                 r_k^old = r_k^old ∪ {τ_ν};
13:                 r_k^full = r_k^full ∪ {τ_ν};
14:             }
15:         }

        /* Let I^Δ be the set {t_1, ..., t_g}. */
16:     for (j = 1; j <= g; j++)
17:         for (k = 1; k <= n; k++)
18:         {
19:             ν = ∅;
20:             if (match(A_k, t_j))
21:             {
22:                 r_k^Δ = r_k^Δ ∪ {τ_ν};
23:                 r_k^full = r_k^full ∪ {τ_ν};
24:             }
25:         }

26:     for (p = 1; p <= n; p++)
27:     {
28:         create a relation r_body = π_{v_1,...,v_q}(r_1^old ⋈ ... ⋈ r_{i-1}^old ⋈ r_i^Δ ⋈ r_{i+1}^full ⋈ ... ⋈ r_n^full)
29:         where {v_1, ..., v_q} is the set of distinct variables in A_0;

            /* Let r_body be the set {u_1, ..., u_w}. */
30:         for (h = 1; h <= w; h++)
31:             I^new = I^new ∪ {A_0 ψ_{u_h}};
32:     }
33: }
```


Figure 4.1: Procedure for seminaive rule application

As for the discussion of naive rule application, the concept of rule transforms (see Definition 7 in Chapter 3) facilitates a proper discussion of the behaviour of the semi_apply function. In particular, given a *HiLog* rule c and instances of the global relations I^{old} and I^{Δ}, the function should *ideally* add to I^{new} all the elements of $T_c(I^{old} \cup I^{\Delta}) - T_c(I^{old})$, and only those elements. In practice, though, the completeness and correctness of seminaive evaluation require only that the function add to I^{new} a set of ground terms which is a (not necessarily proper) superset of $T_c(I^{old} \cup I^{\Delta}) - T_c(I^{old})$ and a (not necessarily proper) subset of $T_c(I^{old} \cup I^{\Delta})$. The three theorems which follow prove that the function does indeed behave as required. The first theorem proves a useful equivalence between two relational algebra expressions; the second defines an expression, $\Lambda_{c,I^{old},I^{\Delta}}$, in terms of c, I^{old} and I^{Δ}, and proves that the value of the expression is bounded by $T_c(I^{old} \cup I^{\Delta}) - T_c(I^{old})$ and $T_c(I^{old} \cup I^{\Delta})$; finally, the third theorem proves that the set of ground terms added to I^{new} by semi_apply is precisely equal to $\Lambda_{c,I^{old},I^{\Delta}}$.

Theorem 11 *Let $n \in N$, $n \geq 1$. For each $i \in N$, $1 \leq i \leq n$, let r_i^{old}, r_i^{Δ} and r_i^{full} be relations over a common relation scheme and let $r_i^{full} = r_i^{old} \cup r_i^{\Delta}$. Then $(r_1^{full} \bowtie \ldots \bowtie r_n^{full}) - (r_1^{old} \bowtie \ldots \bowtie r_n^{old}) = (\bigcup_{i=1}^{n}(r_1^{old} \bowtie \ldots \bowtie r_{i-1}^{old} \bowtie r_i^{\Delta} \bowtie r_{i+1}^{full} \bowtie \ldots \bowtie r_n^{full})) - (r_1^{old} \bowtie \ldots \bowtie r_n^{old})$*

Proof: The proof is an induction on n. Assume that the equality holds for all $n \in N$, $1 \leq n \leq k$. Now consider the case where $n = k + 1$:

$$
\begin{aligned}
&(r_1^{full} \bowtie \ldots \bowtie r_n^{full}) - (r_1^{old} \bowtie \ldots \bowtie r_n^{old}) \\
&= (r_1^{full} \bowtie \ldots \bowtie r_{k+1}^{full}) - (r_1^{old} \bowtie \ldots \bowtie r_{k+1}^{old}) \\
&= ((r_1^{full} \bowtie \ldots \bowtie r_k^{full}) \bowtie r_{k+1}^{full}) - (r_1^{old} \bowtie \ldots \bowtie r_{k+1}^{old}) \\
&= ((((r_1^{full} \bowtie \ldots \bowtie r_k^{full}) - (r_1^{old} \bowtie \ldots \bowtie r_k^{old})) \cup (r_1^{old} \bowtie \ldots \bowtie r_k^{old})) \bowtie r_{k+1}^{full}) \\
&\quad - (r_1^{old} \bowtie \ldots \bowtie r_{k+1}^{old}) \\
&\quad \text{since } (r_1^{old} \bowtie \ldots \bowtie r_k^{old}) \subseteq (r_1^{full} \bowtie \ldots \bowtie r_k^{full}) \\
&= ((((\bigcup_{i=1}^{k}(r_1^{old} \bowtie \ldots \bowtie r_{i-1}^{old} \bowtie r_i^{\Delta} \bowtie r_{i+1}^{full} \bowtie \ldots \bowtie r_k^{full})) - (r_1^{old} \bowtie \ldots \bowtie r_k^{old})) \\
&\quad \cup (r_1^{old} \bowtie \ldots \bowtie r_k^{old})) \bowtie r_{k+1}^{full}) - (r_1^{old} \bowtie \ldots \bowtie r_{k+1}^{old}) \\
&\quad \text{by the inductive hypothesis}
\end{aligned}
$$

$$
\begin{aligned}
&= ((\bigcup_{i=1}^{k}(r_1^{old} \bowtie \ldots \bowtie r_{i-1}^{old} \bowtie r_i^{\Delta} \bowtie r_{i+1}^{full} \bowtie \ldots \bowtie r_k^{full} \bowtie r_{k+1}^{full})) \\
&\quad \cup (r_1^{old} \bowtie \ldots \bowtie r_k^{old} \bowtie (r_{k+1}^{old} \cup r_{k+1}^{\Delta}))) - (r_1^{old} \bowtie \ldots \bowtie r_{k+1}^{old}) \\
&= ((\bigcup_{i=1}^{k}(r_1^{old} \bowtie \ldots \bowtie r_{i-1}^{old} \bowtie r_i^{\Delta} \bowtie r_{i+1}^{full} \bowtie \ldots \bowtie r_{k+1}^{full})) \cup (r_1^{old} \bowtie \ldots \bowtie r_{k+1}^{old}) \\
&\quad \cup (r_1^{old} \bowtie \ldots \bowtie r_k^{old} \bowtie r_{k+1}^{\Delta})) - (r_1^{old} \bowtie \ldots \bowtie r_{k+1}^{old}) \\
&= ((\bigcup_{i=1}^{k}(r_1^{old} \bowtie \ldots \bowtie r_{i-1}^{old} \bowtie r_i^{\Delta} \bowtie r_{i+1}^{full} \bowtie \ldots \bowtie r_{k+1}^{full})) \cup (r_1^{old} \bowtie \ldots \bowtie r_k^{old} \bowtie r_{k+1}^{\Delta})) \\
&\quad - (r_1^{old} \bowtie \ldots \bowtie r_{k+1}^{old}) \\
&= (\bigcup_{i=1}^{k+1}(r_1^{old} \bowtie \ldots \bowtie r_{i-1}^{old} \bowtie r_i^{\Delta} \bowtie r_{i+1}^{full} \bowtie \ldots \bowtie r_{k+1}^{full})) - (r_1^{old} \bowtie \ldots \bowtie r_{k+1}^{old}) \\
&= (\bigcup_{i=1}^{n}(r_1^{old} \bowtie \ldots \bowtie r_{i-1}^{old} \bowtie r_i^{\Delta} \bowtie r_{i+1}^{full} \bowtie \ldots \bowtie r_n^{full})) - (r_1^{old} \bowtie \ldots \bowtie r_n^{old})
\end{aligned}
$$

For the basis, observe that, when $n = 1$,

$$
\begin{aligned}
&(r_1^{full} \bowtie \ldots \bowtie r_n^{full}) - (r_1^{old} \bowtie \ldots \bowtie r_n^{old}) \\
&= r_1^{full} - r_1^{old} \\
&= (r_1^{old} \cup r_1^{\Delta}) - r_1^{old} \\
&= r_1^{\Delta} - r_1^{old}
\end{aligned}
$$

Now, note that, when $i = 1$ the expression $(r_1^{old} \bowtie \ldots \bowtie r_{i-1}^{old} \bowtie r_i^{\Delta} \bowtie r_{i+1}^{full} \bowtie \ldots \bowtie r_n^{full})$ reduces to $(r_1^{\Delta} \bowtie r_2^{full} \bowtie \ldots \bowtie r_n^{full})$. If $n = 1$ the expression reduces further to r_1^{Δ}. Thus, when $n = 1$, $(\bigcup_{i=1}^{n}(r_1^{old} \bowtie \ldots \bowtie r_{i-1}^{old} \bowtie r_i^{\Delta} \bowtie r_{i+1}^{full} \bowtie \ldots \bowtie r_n^{full})) - (r_1^{old} \bowtie \ldots \bowtie r_n^{old})$ is also equal to $r_1^{\Delta} - r_1^{old}$ and the basis is proved. □

Theorem 12 *Let L be a language of HiLog with Herbrand Universe H_L. Let I^{old} and I^{Δ} be elements of $\mathcal{P}(H_L)$. Let c be the definite HiLog Horn clause $A_0 \vee \neg A_1 \vee \ldots \vee \neg A_n$, where $n \in N$, $n \geq 1$ and $A_0, \ldots, A_n$ are nonground HiLog terms. Assume that the set of distinct variable symbols in A_0 is $\{v_1, \ldots, v_q\}$ and that, for each $i \in N$, $1 \leq i \leq n$, the set of distinct variable symbols in A_i is $\{v_{i_1}, \ldots, v_{i_{m_i}}\}$. Let $\Lambda_{c,I^{\mathrm{old}},I^{\Delta}} = \bigcup_{i=1}^{n}\{A_0\mu \mid \tau_\mu \in \pi_{v_1,\ldots,v_q}(r_1^{old} \bowtie \ldots \bowtie r_{i-1}^{old} \bowtie r_i^{\Delta} \bowtie r_{i+1}^{full} \bowtie \ldots \bowtie r_n^{full})\}$, where, for all $i \in N$, $1 \leq i \leq n$, r_i^{old}, r_i^{Δ} and r_i^{full} are defined as follows: each is a relation over the relation scheme $(v_{i_1}, \ldots, v_{i_{m_i}})$ in which each attribute has domain H_L; specifically, $r_i^{old} = \{u_i \in {H_L}^{m_i} \mid A_i\psi_{u_i} \in I^{old}\}$,*

$r_i^{\Delta} = \{u_i \in {H_L}^{m_i} \mid A_i\psi_{u_i} \in I^{\Delta}\}$ *and* $r_i^{full} = r_i^{old} \cup r_i^{\Delta}$. *Then* $T_c(I^{old} \cup I^{\Delta}) - T_c(I^{old}) \subseteq \Lambda_{c,I^{\text{old}},I^{\Delta}} \subseteq T_c(I^{old} \cup I^{\Delta})$.

Proof: By Theorem 6, $T_c(I^{old} \cup I^{\Delta}) = \{A_0\mu \mid \tau_\mu \in \pi_{v_1,\ldots,v_q}(r_1 \bowtie \ldots \bowtie r_n)\}$, where, for all $i \in N$, $1 \leq i \leq n$,

$$r_i = \{u_i \in {H_L}^{m_i} \mid A_i\psi_{u_i} \in (I^{old} \cup I^{\Delta})\}.$$

Note that $r_i = r_i^{full}$, since

$$\begin{aligned}
&\{u_i \in {H_L}^{m_i} \mid A_i\psi_{u_i} \in (I^{old} \cup I^{\Delta})\} = \\
&\qquad \{u_i \in {H_L}^{m_i} \mid A_i\psi_{u_i} \in I^{old}\} \cup \{u_i \in {H_L}^{m_i} \mid A_i\psi_{u_i} \in I^{\Delta}\} \\
&= \quad r_i^{old} \cup r_i^{\Delta} = r_i^{full}.
\end{aligned}$$

Also, by Theorem 6, $T_c(I^{old}) = \{A_0\mu \mid \tau_\mu \in \pi_{v_1,\ldots,v_q}(r_1' \bowtie \ldots \bowtie r_n')\}$, where, for all $i \in N$, $1 \leq i \leq n$,

$$r_i' = \{u_i \in {H_L}^{m_i} \mid A_i\psi_{u_i} \in I^{old}\}.$$

Clearly, $r_i' = r_i^{old}$. Thus it follows that

$$\begin{aligned}
&T_c(I^{old} \cup I^{\Delta}) - T_c(I^{old}) \\
&= \{A_0\mu \mid \tau_\mu \in \pi_{v_1,\ldots,v_q}(r_1^{full} \bowtie \ldots \bowtie r_n^{full})\} - \{A_0\mu \mid \tau_\mu \in \pi_{v_1,\ldots,v_q}(r_1^{old} \bowtie \ldots \bowtie r_n^{old})\} \\
&= \{A_0\mu \mid \tau_\mu \in \pi_{v_1,\ldots,v_q}((r_1^{full} \bowtie \ldots \bowtie r_n^{full}) - (r_1^{old} \bowtie \ldots \bowtie r_n^{old}))\} - T_c(I^{old}) \\
&= \{A_0\mu \mid \tau_\mu \in \pi_{v_1,\ldots,v_q}((\bigcup_{i=1}^{n}(r_1^{old} \bowtie \ldots \bowtie r_{i-1}^{old} \bowtie r_i^{\Delta} \bowtie r_{i+1}^{full} \bowtie \ldots \bowtie r_n^{full})) \\
&\quad - (r_1^{old} \bowtie \ldots \bowtie r_n^{old}))\} - T_c(I^{old}) \\
&\quad \text{by Theorem 11} \\
&= \{A_0\mu \mid \tau_\mu \in \pi_{v_1,\ldots,v_q}(\bigcup_{i=1}^{n}(r_1^{old} \bowtie \ldots \bowtie r_{i-1}^{old} \bowtie r_i^{\Delta} \bowtie r_{i+1}^{full} \bowtie \ldots \bowtie r_n^{full}))\} \\
&\quad - \{A_0\mu \mid \tau_\mu \in \pi_{v_1,\ldots,v_q}(r_1^{old} \bowtie \ldots \bowtie r_n^{old})\} - T_c(I^{old}) \\
&= \bigcup_{i=1}^{n}\{A_0\mu \mid \tau_\mu \in \pi_{v_1,\ldots,v_q}(r_1^{old} \bowtie \ldots \bowtie r_{i-1}^{old} \bowtie r_i^{\Delta} \bowtie r_{i+1}^{full} \bowtie \ldots \bowtie r_n^{full})\} - T_c(I^{old}) \\
&= \Lambda_{c,I^{\text{old}},I^{\Delta}} - T_c(I^{old})
\end{aligned}$$

Then it is clearly true that $\Lambda_{c,I^{\text{old}},I^{\Delta}} \supseteq T_c(I^{old} \cup I^{\Delta}) - T_c(I^{old})$.

Now consider any $t \in \Lambda_{c,I^{\text{old}},I^{\Delta}}$: if $t \in T_c(I^{old})$, then, since $T_c(I^{old})$ is clearly a subset of $T_c(I^{old} \cup I^{\Delta})$, it follows that $t \in T_c(I^{old} \cup I^{\Delta})$. On the other hand, if $t \notin T_c(I^{old})$, then

$t \in (\Lambda_{c,I^{\text{old}},I^{\Delta}} - T_c(I^{old}))$. Since the latter is equal to $T_c(I^{old} \cup I^{\Delta}) - T_c(I^{old})$, it again follows that $t \in T_c(I^{old} \cup I^{\Delta})$. So $\Lambda_{c,I^{\text{old}},I^{\Delta}} \subseteq T_c(I^{old} \cup I^{\Delta})$ and the theorem is proved. □

Theorem 13 *Assume that the function* semi_apply *is invoked with argument c, where c is any definite HiLog Horn clause denoting a HiLog rule, and in the presence of the global sets of HiLog ground terms I^{old}, I^{Δ} and I^{new}. Let J denote the set of HiLog ground terms added to I^{new} by the call to* semi_apply. *Then $T_c(I^{old} \cup I^{\Delta}) - T_c(I^{old}) \subseteq J \subseteq T_c(I^{old} \cup I^{\Delta})$.*

Proof: Let c be a definite *HiLog* Horn clause in the *HiLog* language L, where L has Herbrand Universe H_L. Specifically, let $c = A_0 \vee \neg A_1 \vee \ldots \vee \neg A_n$, where $n \in N$, $n \geq 1$, and $A_0, \ldots, A_n$ are nonground *HiLog* terms. Let the set of distinct variable symbols in A_0 be $\{v_1, \ldots, v_q\}$ and, for each $i \in N$, $1 \leq i \leq n$, let the set of distinct variable symbols in A_i be $\{v_{i_1}, \ldots, v_{i_{m_i}}\}$. Let

$$\Lambda_{c,I^{\text{old}},I^{\Delta}} = \bigcup_{i=1}^{n} \{A_0\mu \mid \tau_\mu \in \pi_{v_1,\ldots,v_q}(s_1^{old} \bowtie \ldots \bowtie s_{i-1}^{old} \bowtie s_i^{\Delta} \bowtie s_{i+1}^{full} \bowtie \ldots \bowtie s_n^{full})\}$$

where, for all $i \in N$, $1 \leq i \leq n$, s_i^{old}, s_i^{Δ} and s_i^{full} are defined as follows: each is a relation over the relation scheme $(v_{i_1}, \ldots, v_{i_{m_i}})$ in which each attribute has domain H_L; specifically:

- $s_i^{old} = \{u_i \in {H_L}^{m_i} \mid A_i\psi_{u_i} \in I^{old}\}$;
- $s_i^{\Delta} = \{u_i \in {H_L}^{m_i} \mid A_i\psi_{u_i} \in I^{\Delta}\}$;
- $s_i^{full} = s_i^{old} \cup s_i^{\Delta}$.

By Theorem 12, $T_c(I^{old} \cup I^{\Delta}) - T_c(I^{old}) \subseteq \Lambda_{c,I^{\text{old}},I^{\Delta}} \subseteq T_c(I^{old} \cup I^{\Delta})$, so it suffices to prove that $J = \Lambda_{c,I^{\text{old}},I^{\Delta}}$.

Observe that the for-loop of lines 3–5 creates, for each $i \in N$, $1 \leq i \leq n$, relations r_i^{old}, r_i^{Δ} and r_i^{full} over the same relation schemes over which s_i^{old}, s_i^{Δ} and s_i^{full} are defined. Now, in the nested for-loop of lines 6–15, the compound statement of lines 8–15 is executed for each t_j in I^{old} and each A_k in the body of c. By Theorems 4 and 5, the match procedure is complete and correct, so line 12 will add τ_ν to r_k^{old} if, and only if, $A_k\nu = t_j$. Since s_k^{old} can be rewritten as $\{\tau_\nu \in {H_L}^{m_k} \mid A_k\nu \in I^{old}\}$, it follows that, after the execution

of the nested for-loop of lines 6–15, $r_i^{old} = s_i^{old}$ for all $i \in N$, $1 \leq i \leq n$. By a similar argument, $r_i^{\Delta} = s_i^{\Delta}$, for all $i \in N$, $1 \leq i \leq n$, after the execution of the nested for-loop of lines 16–25. Furthermore, lines 13 and 23 ensure that, whenever a ground term is added to r_i^{old} or r_i^{Δ}, it is also added to r_i^{full}. Thus, after execution of the nested for-loops, $r_i^{full} = r_i^{old} \cup r_i^{\Delta} = s_i^{old} \cup s_i^{\Delta} = s_i^{full}$, for all $i \in N$, $1 \leq i \leq n$.

Now observe that the value of $\Lambda_{c,I^{\text{old}},I^{\Delta}}$ may be regarded as the union of the values of n expressions, each of the form

$$\{A_0\mu \mid \tau_\mu \in \pi_{v_1,\ldots,v_q}(s_1^{old} \bowtie \ldots \bowtie s_{p-1}^{old} \bowtie s_p^{\Delta} \bowtie s_{p+1}^{full} \bowtie \ldots \bowtie s_n^{full})\}$$

where $p \in N$, $1 \leq p \leq n$. Thus, in order to show that $J = \Lambda_{c,I^{\text{old}},I^{\Delta}}$, it suffices to show that, for each $p \in N$, $1 \leq p \leq n$, the for-loop of lines 26–32 adds to I^{new} the value of

$$\{A_0\mu \mid \tau_\mu \in \pi_{v_1,\ldots,v_q}(s_1^{old} \bowtie \ldots \bowtie s_{p-1}^{old} \bowtie s_p^{\Delta} \bowtie s_{p+1}^{full} \bowtie \ldots \bowtie s_n^{full})\}.$$

Note that, since $r_p^{old} = s_p^{old}$, $r_p^{\Delta} = s_p^{\Delta}$ and $r_p^{full} = s_p^{full}$, the r_{body} relation created on line 28 is equal to $\pi_{v_1,\ldots,v_q}(s_1^{old} \bowtie \ldots \bowtie s_{p-1}^{old} \bowtie s_p^{\Delta} \bowtie s_{p+1}^{full} \bowtie \ldots \bowtie s_n^{full})$. Thus the for-loop of lines 30–31 adds to I^{new} the ground terms of

$$\{A_0\psi_{u_h} \mid u_h \in \pi_{v_1,\ldots,v_q}(s_1^{old} \bowtie \ldots \bowtie s_{p-1}^{old} \bowtie s_p^{\Delta} \bowtie s_{p+1}^{full} \bowtie \ldots \bowtie s_n^{full})\}.$$

This latter expression can clearly be rewritten as

$$\{A_0\mu \mid \tau_\mu \in \pi_{v_1,\ldots,v_q}(s_1^{old} \bowtie \ldots \bowtie s_{p-1}^{old} \bowtie s_p^{\Delta} \bowtie s_{p+1}^{full} \bowtie \ldots \bowtie s_n^{full})\}$$

so $J = \Lambda_{c,I^{\text{old}},I^{\Delta}}$ and the theorem is proved. □

4.2.2 The Seminaive Evaluation Algorithm

Section 4.1 provided an intuitive introduction to the seminaive evaluation algorithm and illustrated its application to a small set of facts and rules. This section presents a more rigorous description of the algorithm, proves its correctness and completeness, and concludes by demonstrating that the algorithm does indeed exhibit the nonrepetition property described in Section 4.1. For discussions of seminaive evaluation in the context of Datalog, refer to [6, 5, 9].

```
1:  least_semi(P)
    /* P is a finite set of definite Hilog Horn clause rules; I^old, I^Δ and
    I^new are global sets of Hilog ground terms. */
2:  {
3:      I^old = ∅;

4:      while (I^new != ∅)
5:      {
6:          I^Δ = I^new;
7:          I^new = ∅;

            /* Let P be the set {c_1, ..., c_m}. */
8:          for (i = 1; i <= m; i++)
9:              semi_apply(c_i);

10:         I^old = I^old ∪ I^Δ;
11:         I^new = I^new − I^old;
12:     }
13: }
```

Figure 4.2: Procedure for seminaive evaluation

Recall that the purpose of the seminaive evaluation algorithm is identical to that of the naive evaluation algorithm described in the previous chapter, namely to compute the least Herbrand model of a set of *HiLog* facts and a *HiLog* program comprising only definite *HiLog* Horn clause rules. However, while naive evaluation will generally repeat derivations when it is applied to a recursive program, seminaive evaluation overcomes this inefficiency by ensuring that no fact is derived by the same means more than once. The algorithm is based on the procedure least_semi (see Figure 4.2) which accepts as an argument a set P of definite *HiLog* Horn clause rules and operates in the presence of three global sets of *HiLog* ground terms: I^{old}, I^{Δ} and I^{new}. Assume that, when least_semi is called, $I^{new} = F$ and that the least Herbrand model M of F and P is finite. Then least_semi terminates with $I^{old} = M$.

Theorem 14 below proves the correctness of the least_semi function.

Theorem 14 *Let M be the least Herbrand model of a set of HiLog ground terms F and a set of HiLog rules P. Assume that the procedure* least_semi *is invoked with P as an*

argument and with I^{new} *set equal to* F. *Then, throughout the execution of* least_semi, I^{old} *remains a subset of* M.

Proof: First consider the special case where $F = \emptyset$. Line 3 sets I^{old} equal to $\emptyset$. Also, since I^{new} is initially equal to F, $I^{new} = \emptyset$ when the condition of line 5 is first tested. Clearly, the condition tests false and so the algorithm terminates immediatedly with $I^{old} = \emptyset$. Since $\emptyset \subseteq M$, the theorem is proved true for this special case.

However, if $F \neq \emptyset$, the while-loop of lines 4–12 is executed at least once. In this case, the proof relies on a demonstration, by induction on the number of iterations of the while-loop, that, whenever the for-loop of lines 8–9 is executed, $(I^{old} \cup I^{\Delta}) \subseteq M$. Assume that the assertion holds true on iteration j of the while-loop, where $j \in N$, $j \geq 1$. Then, for any $c \in P$, it follows from Theorem 8 that $T_c(I^{old} \cup I^{\Delta}) \subseteq M$. Now, by Theorem 13, the set of *HiLog* ground terms added to I^{new} by the call to semi_apply, with argument c, is a subset of $T_c(I^{old} \cup I^{\Delta})$. Thus all the ground terms added to I^{new} are elements of M and, after the execution of the for-loop of lines 8–9, $I^{new} \subseteq M$. Now, line 10 sets I^{old} equal to $I^{old} \cup I^{\Delta}$, which, by the inductive hypothesis, is a subset of M. Thus $I^{old} \subseteq M$ when the for-loop is executed on iteration $(j + 1)$. Furthermore, since I^{new} clearly remains a subset of M after execution of line 11 on iteration j, and since line 6 sets I^{Δ} equal to I^{new} on iteration $(j + 1)$, $I^{\Delta} \subseteq M$ when the for-loop is executed on iteration $(j + 1)$ and the inductive step is proved. For the basis of the induction, observe that, owing to the execution of line 3, $I^{old} = \emptyset$ when the for-loop is executed on the first iteration. Also, since I^{new} is initially equal to F and line 6 sets I^{Δ} equal to I^{new}, $I^{old} \cup I^{\Delta} = I^{\Delta} = F$ when the for-loop is executed on the first iteration. By the definition of "model", $F \subseteq M$ and the basis is proved.

To complete the proof of the theorem, it suffices to note that if, on any given iteration of the while-loop, $(I^{old} \cup I^{\Delta}) \subseteq M$ prior to execution of the for-loop, then, on that iteration of the while-loop, I^{old} is clearly a subset of M both before and after the execution of line 10. □

The following theorem essentially states that any fact which can be derived by applying a rule to the database, but whose derivation is based exclusively on "old facts," must already be present in the database.

Theorem 15 *Assume that the procedure* least_semi *is invoked with argument P, where P is a nonempty set of HiLog rules, and in the presence of the global sets I^{old} and I^{Δ} of HiLog ground terms. Then, on any iteration of the* while*-loop of lines 4–12, the following holds during the execution of the* for*-loop of lines 8–9: for any HiLog ground term t and any $c \in P$, if $t \in T_c(I^{old})$, then either $t \in I^{old}$ or $t \in I^{\Delta}$.*

Proof: The proof is an induction on the number of the iteration of the while-loop. Assume that the theorem holds on any iteration j of the while-loop, where $j \in N$, $1 \leq j \leq n$. Now assume that, on iteration $(n+1)$ of the while-loop, $t \in T_c(I^{old})$ during execution of the for-loop of lines 8–9, where t is a *HiLog* ground term and $c \in P$. If c is the definite *HiLog* Horn clause $A_0 \vee \neg A_1 \vee \ldots \vee \neg A_p$, where $p \in N$, $p \geq 1$ and $A_0, \ldots, A_p$ are nonground *HiLog* terms, then it follows from the definition of T_c that there exists a variable assignment ν under which $A_0\nu = t$ and, for all $k \in N$, $1 \leq k \leq p$, $A_k\nu \in I^{old}$. Now a consideration of line 10 of the algorithm reveals that every element of I^{old} was, at one stage, an element of I^{Δ}, so that, during the execution of the for-loop on iteration n of the while-loop, each $A_j\nu$ was either an element of I^{old} or an element of I^{Δ}. Clearly, each was an element of $(I^{old} \cup I^{\Delta})$, so that t was necessarily an element of $T_c(I^{old} \cup I^{\Delta})$ during execution of the for-loop on iteration n. If t was also an element of $T_c(I^{old})$, then, by the inductive hypothesis, it was an element of $(I^{old} \cup I^{\Delta})$ on iteration n and, owing to line 10 of the algorithm, is clearly an element of I^{old} on iteration $(n+1)$. Otherwise, t was an element of $T_c(I^{old} \cup I^{\Delta}) - T_c(I^{old})$ and, by Theorem 13, was added to I^{new} by the for-loop on iteration n of the while-loop. Then, when line 11 was executed on iteration n of the while-loop, either t was an element of I^{old}, in which case it remains an element of I^{old} on iteration $(n+1)$, or t remained an element of I^{new}, in which case it was added to I^{Δ} by line 6 on iteration $(n+1)$ of the while-loop. It follows that, when the for-loop is executed on iteration $(n+1)$ of the while-loop, $t \in I^{old}$ or $t \in I^{\Delta}$, so the inductive step is proved.

For the basis of the induction, observe that, when the for-loop is executed on the first iteration of the while-loop, $I^{old} = \emptyset$. Thus, for any $c \in P$, $T_c(I^{old}) = \emptyset$ and, since the empty set is clearly a subset of I^{old} and of I^{Δ}, the basis is proved. □

Theorem 16 below uses Theorem 15 to prove that the least_semi function is complete.

Theorem 16 *Let F be a set of HiLog ground terms, let P be a set of HiLog rules and assume that the least Herbrand model M of F and P is finite. Then, if* least_semi *is invoked with P as an argument and with $I^{new} = F$, execution of the procedure will terminate with $I^{old} = M$.*

Proof: First consider the special case where $F = \emptyset$. Line 3 sets I^{old} equal to $\emptyset$. Also, since I^{new} is initially equal to F, $I^{new} = \emptyset$ when the condition of line 4 is first tested. Clearly the condition tests false and so the algorithm terminates immediately with $I^{old} = \emptyset$. Since the least Herbrand model of a program and an empty set of facts is simply $\emptyset$, the algorithm performs correctly in this case.

More generally, assume that $F \neq \emptyset$. The subtraction on line 11 and the condition of line 4 ensure that the body of the while-loop is executed only if I^{new} is a nonempty set of ground terms with $I^{old} \cap I^{new} = \emptyset$. Since line 6 sets I^{Δ} equal to I^{new}, and line 10, in turn, adds the elements of I^{Δ} to I^{old}, it follows that the number of elements in I^{old} increases on each iteration of the while-loop. Now, by Theorem 14, I^{old} remains a subset of M, so, if M is finite, the while-loop cannot be repeated indefinitely and the algorithm must eventually terminate. Since I^{old} remains a subset of M, it suffices to show that, when the algorithm terminates, I^{old} is a model of F and P.

Assume that, when least_semi terminates, I^{old} is not a model of F and P. Now observe that, since I^{new} was initially equal to F, where F is a nonempty set of *HiLog* ground terms, the body of the while-loop must have been executed at least once. Specifically, on the first iteration of the while-loop, line 6 set I^{Δ} equal to F and line 10, in turn, added all the elements of F to I^{old}, so that $F \subseteq I^{old}$. Thus, if I^{old} is not a model of F and P, it must be because I^{old} fails to satisfy at least one of the rules of P. Let c be such a rule and let c be denoted by the definite *HiLog* Horn clause $A_0 \vee \neg A_1 \vee \ldots \vee \neg A_n$, where $n \in N$, $n \geq 1$ and $A_0, \ldots, A_n$ are nonground *HiLog* terms. Then there exists a variable assignment ν under which $A_1\nu, \ldots, A_n\nu$ are all elements of I^{old} and $A_0\nu \notin I^{old}$. Let j be the number of the last iteration of the while-loop and note that, when the for-loop was executed on iteration j of the while-loop, each of the ground terms $A_1\nu, \ldots, A_n\nu$ must have been elements of $I^{old} \cup I^{\Delta}$. Thus $A_0\nu$ must have been an element of $T_c(I^{old} \cup I^{\Delta})$. Furthermore $A_0\nu$ could not have been an element of $T_c(I^{old})$, since, by Theorem 15, it would then have been

an element of $(I^{old} \cup I^{\Delta})$ and would thus have been an element of I^{old} after execution of line 10. Clearly, therefore, $A_0\nu$ was an element of $T_c(I^{old} \cup I^{\Delta}) - T_c(I^{old})$ when the for-loop was executed on iteration j of the while-loop, and, by Theorem 13, was thus added to I^{new} by the execution of the for-loop. Also, since $A_0\nu$ could not have been an element of I^{old} when line 11 was executed, $A_0\nu$ must have remained an element of I^{new} at the end of iteration j of the while-loop. But then the condition of line 4 would have tested true after execution of iteration j, and so iteration j could not have been the final iteration of the while-loop. The contradiction forces the conclusion that, when least_semi terminates, I^{old} is indeed a model of F and P. According to the definition of "least model," $M \subseteq I^{old}$ and, by Theorem 14, $I^{old} \subseteq M$. So $I^{old} = M$. □

Theorem 17 *Seminaive evaluation, as implemented by the* semi_apply *and* least_semi *functions, has the non-repetition property.*

Proof: Consider any derivation performed during the application of a rule, say c, on a given iteration of seminaive evaluation. Line 10 of the least_semi function ensures that, when c is applied on any *subsequent* iteration of seminaive evaluation, all the facts involved in the original derivation are elements of I^{old}. Furthermore, the subtraction on line 11 of least_semi and the assignment on line 6 of least_semi ensure that I^{old} and I^{Δ} are always disjoint when a rule is applied during seminaive evaluation. Thus, if the derivation were repeated on a subsequent iteration of seminaive evaluation, it would be based exclusively on facts in I^{old}. However, seminaive rule application specifically avoids derivations based exclusively on old facts, so the derivation cannot possibly be repeated. □

4.3 The semi System

The semi system is a modified version of the proto system described in Section 3.3 and is based on the simple seminaive evaluation algorithm described in this chapter.

4.3.1 System Organization

Only the following components differ from their counterparts in the proto system:

Pre-evaluation component As in the proto system, this component prepares the data-structure representation of the input program before execution of the evaluation component. Specifically, it prepares an "SQL template" which is used by the evaluation component to generate an SQL statement for each of the relational algebra expressions evaluated on line 28 of semi_apply (Figure 4.1).

Evaluation component The component implements the simple seminaive evaluation algorithm described in this chapter and is based on semi_apply (Figure 4.1) and least_semi (Figure 4.2). It uses the SQL templates generated by the pre-evaluation component to prepare the necessary SQL statements and submits them to the interface provided by the RDBMS platform.

4.3.2 Database Usage

The system uses one single-column table to represent each of the three ground term sets I^{old}, I^{Δ} and I^{new} required by the evaluation algorithm. During application of a rule, it is also required to use three tables, whose columns correspond to subgoal variables, for each of the rule subgoals. These tables correspond to the r_i^{old}, r_i^{Δ} and r_i^{full} relations described on line 4 of semi_apply (see Figure 4.1). String values are used to represent *HiLog* ground terms.

4.4 Performance Analysis: Naive versus Seminaive Evaluation

Since seminaive evaluation endeavours to improve on the efficiency of naive evaluation by eliminating repeated derivations, it seems reasonable to select, as the basis of a comparison of the efficiencies of the procedures, the number of derivations performed by each procedure during evaluation of the least model of a common *HiLog* program. In practice, performing a derivation entails joining tuples drawn from database relations and inserting an appropriate representation of the derived fact into a database relation, so the number of I/O operations which an underlying DBMS executes during computation of a

program's least model may safely be assumed to be an increasing function of the number of derivations performed.

This section uses two analytical examples to demonstrate that, when naive and seminaive evaluation are applied to a common set of facts and rules, the total number of derivations performed by naive evaluation can exceed the total number of derivations performed by seminaive evaluation by a factor which is directly proportional to the number of iterations required by each procedure. It furthermore describes experimental results which suggest that this relationship tends to hold even when the number of new derivations performed on each iteration is essentially random and an exact analysis of the program evaluation is impossible.

Example 11 Consider applying naive and seminaive evaluation to the computation of the least model of a given set of facts and rules and assume that the set of *new* derivations performed by naive evaluation on each iteration is identical to that performed by seminaive evaluation on the corresponding iteration, so that each evaluation requires the same number n of iterations, where $n \in N$ and $n \geq 1$. Assume, furthermore, that the number of new derivations performed on each iteration is a constant d, where $d \in N$, $d \geq 1$.

Clearly, D_{semi}, the total number of derivations performed by seminaive evaluation, is just dn. However, since each iteration of naive evaluation repeats all the derivations performed by all the preceding iterations, D_{naive}, the total number of derivations performed by naive evaluation, is computed as follows:

$$\begin{aligned} D_{naive} &= d + 2d + \cdots + nd \\ &= \sum_{i=1}^{n} di \\ &= d\sum_{i=1}^{n} i \\ &= \frac{dn(n+1)}{2} \end{aligned}$$

Then

$$\begin{aligned} \frac{D_{naive}}{D_{semi}} &= \frac{\frac{dn(n+1)}{2}}{dn} \\ &= \frac{n+1}{2} \end{aligned}$$

Hence the number of derivations performed by naive evaluation exceeds the number of derivations performed by seminaive evaluation by a factor which is proportional to the number of iterations required by each procedure. □

The following example shows that the relationship obtained in Example 11 can also be observed in model computations in which the number of derivations per iteration is not constant.

Example 12 Let F be a set of facts describing a directed graph which constitutes a full binary tree of height n, where $n \in N$, $n \geq 1$, and let P be a pair of rules describing paths in the graph in terms of edge facts:

$$\begin{aligned} r_1\text{:}\quad & p(X,Z) \quad :\!- \quad e(X,Y), p(Y,Z) \\ r_2\text{:}\quad & p(X,Y) \quad :\!- \quad e(X,Y) \end{aligned}$$

To obtain formulae for the total number of derivations performed by naive and seminaive evaluation during computation of the least model of F and P, note that, in each evaluation, the new facts derived on iteration i, where $i \in N$, $1 \leq i \leq n$, correspond to all paths of length i and that the number of such paths is equal to the number of vertices i or more edges from the root, which is equal to $2^{n+1} - 2^i$. So, for seminaive evaluation:

$$\begin{aligned} &\text{Total inferences, } D_{semi}, \\ &= \sum_{i=1}^{n} (2^{n+1} - 2^i) \\ &= n2^{n+1} - 2^{n+1} + 2 \\ &= (n-1)2^{n+1} + 2 \\ &= 2((n-1)2^n + 1) \end{aligned}$$

For naive evaluation, the repetition of derivations ensures that those derivations which are performed for the first time on iteration i will be performed a total of $n + 2 - i$ times, so:

$$\begin{aligned} &\text{Total inferences, } D_{naive}, \\ &= \sum_{i=1}^{n} ((n+2) - i)(2^{n+1} - 2^i) \end{aligned}$$

$$
\begin{aligned}
&= \sum_{i=1}^{n}((n+2)2^{n+1} - i2^{n+1} - (n+2)2^i + i2^i) \\
&= n(n+2)2^{n+1} - 2^{n+1}\sum_{i=1}^{n} i - (n+2)\sum_{i=1}^{n} 2^i + \sum_{i=1}^{n} i2^i \\
&= n(n+2)2^{n+1} - \left(\frac{n(n+1)}{2}\right)2^{n+1} - (n+2)(2^{n+1}-2) + (n-1)2^{n+1} + 2 \\
&= \left(n(n+2) - \frac{n(n+1)}{2} - (n+2) + (n-1)\right)2^{n+1} + 2(n+2) + 2 \\
&= (n^2+3n-6)2^n + 2n + 6
\end{aligned}
$$

Now we propose that, as n approaches infinity, $\frac{D_{naive}}{D_{semi}}$ approaches a linear function with gradient $\frac{1}{2}$. In other words, we seek to prove that

$$\lim_{n\to\infty}\left(\frac{D_{naive}}{D_{semi}} - \frac{n}{2}\right) = k$$

where k is a constant.

The proof is as follows:

$$
\begin{aligned}
&\lim_{n\to\infty}\frac{D_{naive}}{D_{semi}} \\
&= \lim_{n\to\infty}\left(\frac{(n^2+3n-6)2^n + 2n + 6}{2((n-1)2^n+1)} - \frac{n}{2}\right) \\
&= \lim_{n\to\infty}\frac{(n^2+3n-6)2^n + 2n + 6 - n(n-1)2^n - n}{2((n-1)2^n+1)} \\
&= \frac{1}{2}\lim_{n\to\infty}\frac{(n^2+3n-6-n^2+n)2^n + n + 6}{(n-1)2^n+1} \\
&= \frac{1}{2}\lim_{n\to\infty}\frac{(4n-6)2^n + n + 6}{(n-1)2^n+1} \\
&= \frac{1}{2}\lim_{n\to\infty}\frac{(4n-6)2^n\ln 2 + (4)(2^n) + 1}{(n-1)2^n\ln 2 + 2^n} \\
&\quad \text{(l'Hopital's rule)} \\
&= \frac{1}{2}\lim_{n\to\infty}\frac{((4n-6)\ln 2 + 4)2^n + 1}{((n-1)\ln 2 + 1)2^n} \\
&= \frac{1}{2}\lim_{n\to\infty}\frac{((4n-6)\ln 2 + 4)2^n\ln 2 + (4)2^n\ln 2}{((n-1)\ln 2 + 1)2^n\ln 2 + 2^n\ln 2} \\
&\quad \text{(l'Hopital's rule)} \\
&= \frac{1}{2}\lim_{n\to\infty}\frac{(4n-6)\ln 2 + 8}{(n-1)\ln 2 + 2} \\
&= \frac{1}{2}\lim_{n\to\infty}\frac{4\ln 2}{\ln 2} \\
&\quad \text{(l'Hopital's rule)}
\end{aligned}
$$

Iterations	D_{naive}/D_{semi}
5	4.1
8	5.4
8	4.7
8	5.1
4	3
9	5.8
11	7.4
10	6.8

Table 4.1: Seminaive vs Naive evaluation

$= 2$

Thus, if the full-tree graph is sufficiently large, seminaive evaluation will outperform naive evaluation by a factor proportional to the number of iterations required to compute the models. □

It may be supposed that the relationship between $\frac{D_{naive}}{D_{semi}}$ and the number of iterations required to compute a model can hold even when the program is not carefully tailored so that the computation lends itself to a rigorous analysis. Experiment 1 presents empirical evidence that this is indeed the case.

Experiment 1 Each entry in Table 4.1 is based on an input program comprising the rules of Example 12 and a set of "edge facts" describing a directed graph generated by removing edges at random from a fully-connected graph having a random number of vertices. The values of the $\frac{D_{naive}}{D_{semi}}$ ratio were derived from statistics reported by the proto and semi systems.

The plot of this data in Figure 4.3 does indeed suggest a linear relationship between $\frac{D_{naive}}{D_{semi}}$ and the number of iterations.

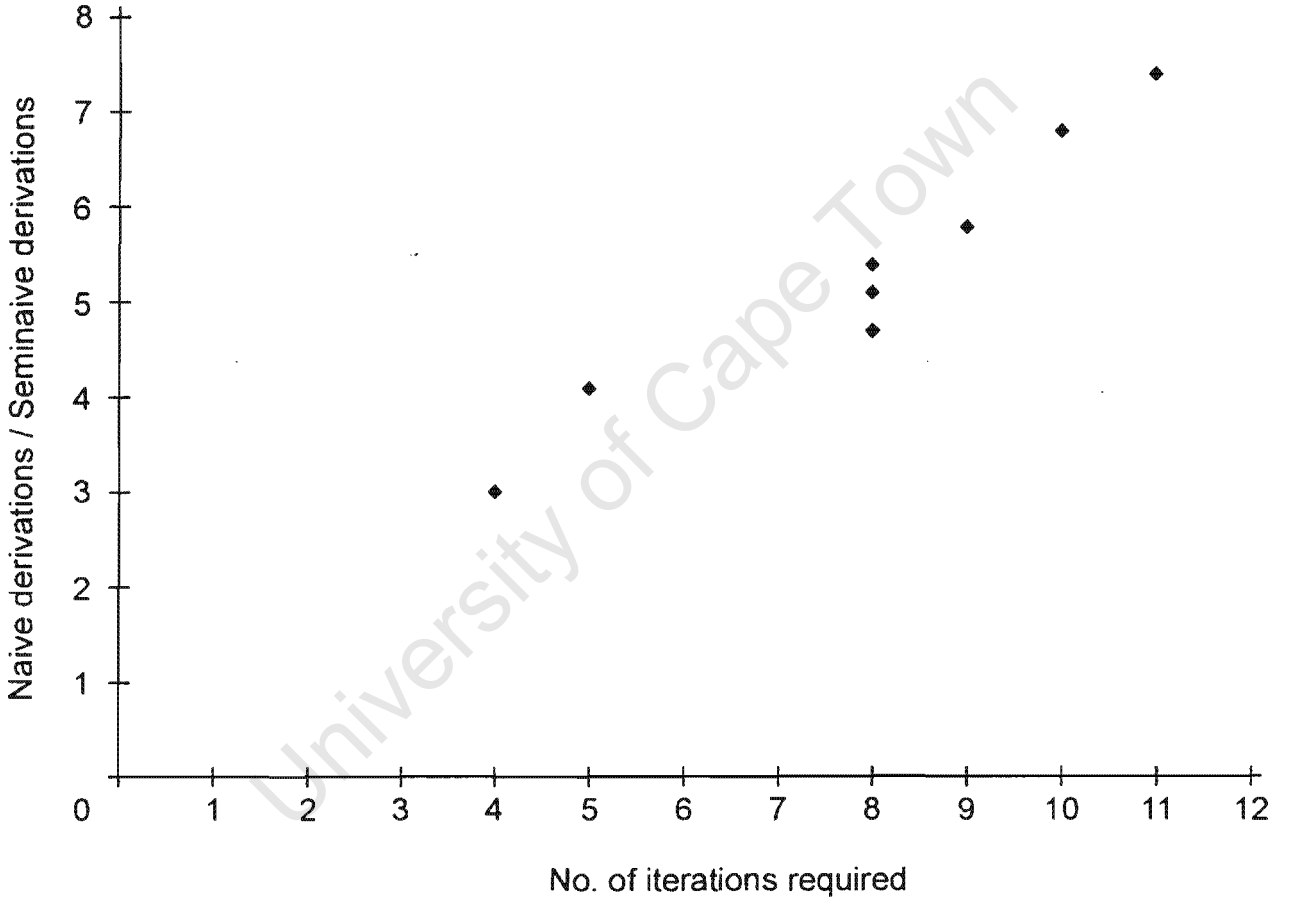


Figure 4.3: Naive/Seminaive vs No. of Iterations

Chapter 5

SCC-based Seminaive Evaluation

The seminaive evaluation algorithm described in the previous chapter represents a substantial improvement over the naive evaluation algorithm presented in Chapter 3. This chapter describes an algorithm, for computing the least Herbrand model of a given set of *HiLog* facts and rules, which attempts to improve still further the efficiency of bottom-up evaluation of *HiLog*.

Section 5.1 uses an example to illustrate some of the shortcomings of conventional seminaive evaluation, and to suggest a means of overcoming these shortcomings by examining *dependencies* amongst the rules of a program, as described for Datalog evaluation in [34, 23, 12]. It also states informally the objectives of the algorithm described in this chapter. Section 5.2 formalizes the notion of "rule dependency," presents several useful definitions and, with the assistance of an example, provides an intuitive overview of the algorithm. Section 5.3 describes the algorithm formally and verifies its correctness and completeness. Section 5.4 describes the sccs system, an enhanced version of the semi system (Section 4.3) capable of identifying and exploiting rule dependencies to improve evaluation efficiency. The chapter concludes by comparing the SCC-based evaluation with the "simple" seminaive evaluation algorithm of the previous chapter, using both formal arguments and data reported by the execution of the semi and sccs systems.

5.1 Motivation and Objectives

Example 9 of the previous chapter detailed the steps involved in applying the conventional seminaive evaluation algorithm to the simple program below, which defines paths in a directed graph in terms of the graph's edge relation:

$$\begin{aligned} r_1\text{:}\quad & p(X,Y) \quad :\!- \quad e(X,Y) \\ r_2\text{:}\quad & p(X,Z) \quad :\!- \quad e(X,Y), p(Y,Z) \end{aligned}$$

It is apparent from the example that only the *first* application of r_1 produces new facts, an observation which is not surprising, since r_1 can only "use" facts which have e as the functor term and neither r_1 nor r_2 produces such facts. It can be said that r_1 is dependent only on the initial database of edge relation facts and not on r_2 or on itself. Thus it is possible to compute the model of the rules and a given database more efficiently by first applying r_1 once, and then applying the seminaive evaluation algorithm to r_2 *alone*. The example below discusses in greater detail this idea of applying rules selectively on the basis of their dependencies.

Example 13 Let F be a set of facts comprising the single element $p(a,b,c,d,e)$ and let P be a program defined in terms of the three rules r_1, r_2 and r_3 below:

$$\begin{aligned} r_1\text{:}\quad & p(E,A,B,C,D) \quad :\!- \quad p(A,B,C,D,E) \\ r_2\text{:}\quad & q(A,B,C) \quad :\!- \quad p(A,B,C,D,E) \\ r_3\text{:}\quad & q(C,A,B) \quad :\!- \quad q(A,B,C) \end{aligned}$$

Figure 5.1 summarizes the steps involved in computing the least Herbrand model of F and P using the seminaive evaluation algorithm of the previous chapter. An examination of the figure reveals several flaws in this straightforward approach and suggests that there is ample scope for improving the efficiency of the evaluation.

First note that repeated application of r_1 to $\{p(a,b,c,d,e)\}$ can yield only four new facts and that no other rule produces facts which can be used by r_1. Yet r_1 is applied on *all eight* iterations of the evaluation, even though the last fact which can be derived by applying the rule is generated on the *fourth* iteration. Similarly, note that r_2 can produce new facts only by using facts which have p as the functor term. Since the last such fact

Database prior to evaluation = $\{p(a, b, c, d, e)\}$

Iteration	Rule Applied	New Facts Generated
1	r_1	$p(e, a, b, c, d)$
	r_2	$q(a, b, c)$
	r_3	—
2	r_1	$p(d, e, a, b, c)$
	r_2	$q(e, a, b)$
	r_3	$q(c, a, b)$
3	r_1	$p(c, d, e, a, b)$
	r_2	$q(d, e, a)$
	r_3	$q(b, e, a),\ q(b, c, a)$
4	r_1	$p(b, c, d, e, a)$
	r_2	$q(c, d, e)$
	r_3	$q(a, d, e),\ q(a, b, e)$
5	r_1	—
	r_2	$q(b, c, d)$
	r_3	$q(e, c, d),\ q(e, a, d)$
6	r_1	—
	r_2	—
	r_3	$q(d, b, c),\ q(d, e, c)$
7	r_1	—
	r_2	—
	r_3	$q(c, d, b)$
8	r_1	—
	r_2	—
	r_3	—

Figure 5.1: Conventional Seminaive Evaluation

is generated on the fourth iteration, the rule need not be applied after the fifth iteration. Furthermore, it is possible to delay the application of r_2 until all the facts which the rule needs have been computed by application of r_1, in which case only *one* application of the rule is necessary. Nevertheless, r_2 is also applied on each of the eight iterations of seminaive evaluation. Finally, consider r_3. Here too it is possible to delay application of the rule until all the facts which it needs have been produced by r_2. Then, since the repeated application of r_3 can clearly generate only two additional facts from each fact produced by r_2, two applications of r_3 should be sufficient to complete computation of the model. However, like r_1 and r_2, r_3 is applied no fewer than *eight* times during the evaluation.

The above discussion suggests the following alternative approach to computing the least Herbrand model of F and P:

- repeatedly apply r_1 until no new facts are generated;
- apply r_2 once;
- repeatedly apply r_3 until no new facts are generated.

Figure 5.2 illustrates how this may be accomplished by first applying seminaive evaluation to r_1 alone, then applying r_2 once and, finally, applying seminaive evaluation to r_3 alone. The figure also demonstrates that this second approach to the evaluation avoids the inefficiencies of straightforward seminaive evaluation:

- r_1 is applied only five times, rather than eight times; the fifth application of the rule is necessary only to verify that application of seminaive evaluation to the single rule is complete—it is the only application of r_1 which does not produce any new facts;
- application of r_2 is delayed until after the application of seminaive evaluation to r_1; this ensures that r_2 need be applied only once, to the complete set of facts which can be produced by r_1, rather than several times, to smaller sets of facts made available to r_2 in a "piecemeal" fashion;
- similarly, the application of r_3 is delayed until all the facts which the rule can use are present in the database; thus, only two applications of r_3 are necessary to ensure

that all those facts which may be derived by applying this rule are duly computed—the third application of r_3 is required only to verify that application of seminaive evaluation to the single rule is complete.

□

Example 13 illustrated that it is possible to significantly improve the efficiency of evaluation by first undertaking a careful study of the *dependencies* amongst rules, in terms of the ability of one rule to generate facts which may be used by another. The remainder of this chapter develops an algorithm, for computing the least Herbrand model of a given set of facts and rules, which exploits a knowledge of such rule dependencies to meet the following objectives:

- insofar as it is possible to do so, avoid applying a rule when its application cannot conceivably produce any new facts;
- delay the application of a rule until as many as possible of the facts which it can use are present in the database.

Meeting these objectives in the context of a practical *HiLog* evaluator based on the semi_apply rule application procedure of Figure 4.1 can be expected to improve the efficiency of the evaluator because it will reduce the total number of calls to semi_apply. Note that the procedure always scans the entire I^{old} and I^{Δ} relations and thus contributes substantially performance cost, even if it doesn't generate any new facts.

5.2 Background Definitions and Overview

This section begins by defining and illustrating several important concepts which facilitate the formal discussion of the algorithm in the succeeding section and concludes with an informal overview of the algorithm. The reader is referred to [34, 23, 12] for discussions of rule-dependencies in the context of Datalog.

Database prior to evaluation = $\{p(a,b,c,d,e)\}$

- Application of seminaive evaluation to r_1:

Iteration	New facts generated
1	$p(e,a,b,c,d)$
2	$p(d,e,a,b,c)$
3	$p(c,d,e,a,b)$
4	$p(b,c,d,e,a)$
5	—

Database subsequent to application of seminaive evaluation to r_1 =

$\{p(a,b,c,d,e), p(e,a,b,c,d), p(d,e,a,b,c), p(c,d,e,a,b), p(b,c,d,e,a)\}$

- Single application of r_2:

Database subsequent to application of rule =

$\{p(a,b,c,d,e), p(e,a,b,c,d), p(d,e,a,b,c), p(c,d,e,a,b), p(b,c,d,e,a),$

$q(a,b,c), q(e,a,b), q(d,e,a), q(c,d,e), q(b,c,d)\}$

- Application of seminaive evaluation to r_3:

Iteration	New facts generated
1	$q(c,a,b)$, $q(b,e,a)$, $q(a,d,e)$, $q(e,c,d)$, $q(d,b,c)$
2	$q(b,c,a)$, $q(a,b,e)$, $q(e,a,d)$, $q(d,e,c)$, $q(c,d,b)$
3	—

Figure 5.2: Seminaive Evaluation based on Rule Dependencies

Definition 13 (Unifiability of *HiLog* terms) *A pair of HiLog terms t_1 and t_2 are said to be* unifiable *if, and only if, there exist* independent *variable assignments ν_1 and ν_2, over the variables of t_1 and t_2 respectively, s.t. $t_1\nu_1$ is identical to $t_2\nu_2$.* □

Example 14 Let $t_1 = f(X, a)$ and let $t_2 = f(b, X)$. Then t_1 and t_2 are clearly unifiable, since, if $\nu_1 = \{(X, b)\}$ and $\nu_2 = \{(X, a)\}$, $t_1\nu_1$ is identical to $t_2\nu_2$.

Now let $t_1 = f(X, Y, Z)$ and let $t_2 = g(X, Y(Z))$. These terms are obviously *not* unifiable, since they will always differ, both in their functor terms and in their arities, no matter what is substituted for their variables.

For a more subtle example, consider the case where $t_1 = f(X)(Y, g(Y))$ and $t_2 = X(Z, Z)$. First note that it is possible to find variable assignments ν_1 and ν_2 under which the functor terms of $t_1\nu_1$ and $t_2\nu_2$ are identical—ν_1 might bind X to, say, q, in which case it would suffice to ensure that ν_2 bound X to $f(q)$. Nevertheless, t_1 and t_2 are not unifiable because, no matter what binding ν_2 contains for Z, the arguments of $t_2\nu_2$ will always be identical, while no variable assignment ν_1 can possibly make the arguments of $t_1\nu_1$ identical. □

A more rigorous and more general treatment of unification, together with an algorithm for unifying first-order logic formulas, may be found in [21]. With little adaptation, the discussion may be applied to the unification of *HiLog* terms.

Now consider two *HiLog* Horn clause rules r_1 and r_2. Intuitively, r_2 is dependent on r_1 if the production of a fact by r_1 can lead, directly or indirectly, to the production of a fact by r_2. In particular, r_2 is *directly dependent* on r_1 if r_1 is able to produce facts which r_2 can use to generate new facts. Clearly this is true only if it is possible to assign ground terms to the variables in the head, h, of r_1 and obtain a ground term which matches at least one subgoal, say s, in the body of r_2. But if this is the case, then there must exist variable assignments μ and ν s.t. $h\mu$ and $s\nu$ are identical—in other words, h and s must be unifiable. This leads to the following definition:

Definition 14 (Direct Dependence of *HiLog* rules) *Let r_1 and r_2 be definite HiLog Horn clause rules. Then r_2 is said to be* directly dependent *on r_1 if, and only if, the head of r_1 unifies with at least one subgoal in the body of r_2.* □

Note that it is also possible for one rule to be *indirectly dependent* on another. Assume, for example, that r_2 is directly dependent on r_1 and that r_3, in turn, is known to be dependent on r_2, in that the production of a fact by r_2 can lead to the production of facts by r_3. Now assume that r_1 is applied to a given database of facts and that the facts produced by the application are then used by r_2 to produce a further set of facts. Since the production of this latter set of facts can lead to the production of yet more facts by r_3, it follows that facts produced by r_1 can lead *indirectly* to the production of facts by r_3. Hence r_3 must be dependent on r_1. These observations lead to the following definition:

Definition 15 (Dependence of *HiLog* rules) *Let r_1 and r_2 be definite HiLog Horn clause rules in a HiLog program P. Then r_2 is said to be* dependent *on r_1 if and only if:*

1. *r_2 is* directly dependent *on r_1, or*
2. *P contains a definite HiLog Horn clause rule r' s.t. r' is directly dependent on r_1 and r_2, in turn, is dependent on r'.*

□

An important feature of a deductive database system based on the *HiLog* language is that it provides support for recursive definitions of functors. The inclusion of such recursive definitions in a program leads to *mutual dependencies* amongst the rules of the program.

Definition 16 (Mutual Dependence of *HiLog* Rules) *Let r_1 and r_2 be definite HiLog Horn clause rules. Then r_1 and r_2 are said to be* mutually dependent *if, and only if, r_1 is dependent on r_2 and r_2, in turn, is dependent on r_1.* □

To illustrate concepts such as direct dependence, indirect dependence and mutual dependence of rules, and to facilitate a discussion of an evaluation algorithm which exploits such dependencies, it is useful to introduce the notion of a rule-dependence graph.

Definition 17 (Rule-dependence Graph of a Program) *Let P be a finite set of definite HiLog Horn clause rules defining a HiLog program. The rule-dependence graph of P, G_P, is defined in terms of the ordered pair (P, E), where $E = \{(r_1, r_2) \in (P \times P) \mid r_2$ is directly dependent on $r_1\}$.* □

Example 15 Consider the following *HiLog* program:

$$
\begin{array}{lrcl}
r_1\text{:} & g(D,A,B,C) & \text{:-} & f(A,B,C,D) \\
r_2\text{:} & f(A,B,C,D) & \text{:-} & g(A,B,C,D) \\
 & & & \\
r_3\text{:} & h(A)(B,C,D) & \text{:-} & f(A,B,C,D), i(A)(B,C,D) \\
r_4\text{:} & i(B)(C,D,A) & \text{:-} & h(A)(B,C,D) \\
 & & & \\
r_5\text{:} & h(A,B)(C,D) & \text{:-} & f(A,B,C,D), j(A,B)(C,D) \\
r_6\text{:} & i(B,C)(D,A) & \text{:-} & h(A,B)(C,D) \\
r_7\text{:} & j(A,B)(C,D) & \text{:-} & i(A,B)(C,D) \\
 & & & \\
r_8\text{:} & k(A(B),C(D)) & \text{:-} & i(A)(B,C,D), n(A,B,C,D) \\
r_9\text{:} & A(B(C))(D) & \text{:-} & k(A(B),C(D)), m(A,B,C,D) \\
r_{10}\text{:} & m(B,C,D,A) & \text{:-} & A(B(C))(D) \\
r_{11}\text{:} & n(A,B,C,D) & \text{:-} & m(A,B,C,D), i(A,B)(C,D)
\end{array}
$$

Figure 5.3 explains the construction of the rule-dependence graph of the program by listing unifiable pairs of rule-heads and subgoals, and the direct dependencies amongst the rules of the program. Figure 5.4 presents a pictorial representation of the graph. □

Now it is intuitively obvious, and straightforward to prove, that, given a pair of rules r_1 and r_2 in a *HiLog* program P, r_2 is dependent on r_1 if, and only if, G_P contains a path from r_1 to r_2. Furthermore, r_1 and r_2 can only be *mutually dependent* if G_P contains both a path from r_1 to r_2 and a path from r_2 to r_1, so that r_1 and r_2 occur in the vertex set of some strongly-connected component (SCC) of G_P. It follows that the vertex sets of the SCCs of G_P represent maximal sets of mutually dependent rules in P. These sets are of interest because they contain rules which cannot be applied in isolation of one another when computing the least Herbrand model of P and some initial database of facts. To guarantee the completeness of the computation, it is necessary to apply the rules of each such set repeatedly until no further facts are generated, i.e. it is necessary to apply a fixpoint evaluation algorithm, like seminaive evaluation, to each set. Henceforth, the term "SCC" will be used to denote both a subgraph of a rule-dependence graph and the vertex set of that subgraph. It will be clear from the context which meaning is intended.

Rule	Rule head	Subgoals unifying with head	Rules containing subgoal	Graph edge
r_1	g(D, A, B, C)	g(A, B, C, D)	r_2	(r_1, r_2)
r_2	f(A, B, C, D)	f(A, B, C, D)	r_1	(r_2, r_1)
			r_3	(r_2, r_3)
			r_5	(r_2, r_5)
r_3	h(A)(B, C, D)	h(A)(B, C, D)	r_4	(r_3, r_4)
r_4	i(B)(C, D, A)	i(A)(B, C, D)	r_3	(r_4, r_3)
			r_8	(r_4, r_8)
r_5	h(A, B)(C, D)	h(A, B)(C, D)	r_6	(r_5, r_6)
r_6	i(B, C)(D, A)	i(A, B)(C, D)	r_7	(r_6, r_7)
			r_{11}	(r_6, r_{11})
r_7	j(A, B)(C, D)	j(A, B)(C, D)	r_5	(r_7, r_5)
r_8	k(A(B), C(D))	k(A(B), C(D))	r_9	(r_8, r_9)
r_9	A(B(C))(D)	A(B(C))(D)	r_{10}	(r_9, r_{10})
r_{10}	m(B, C, D, A)	m(A, B, C, D)	r_9	(r_{10}, r_9)
			r_{11}	(r_{10}, r_{11})
r_{11}	n(A, B, C, D)	n(A, B, C, D)	r_8	(r_{11}, r_8)

Figure 5.3: Construction of Graph for Example Program

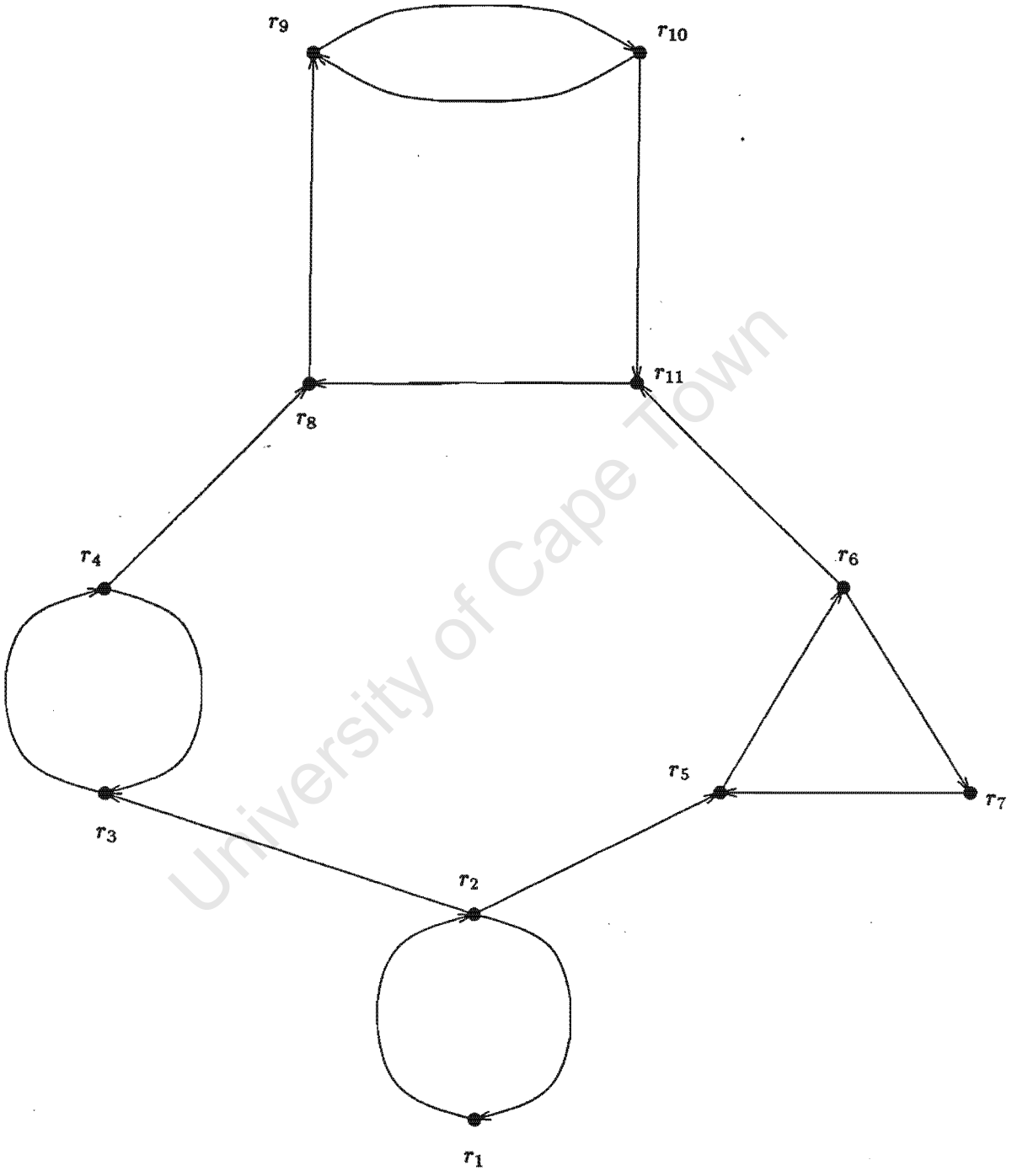


Figure 5.4: Rule Dependence Graph for Program of Example 15

Example 16 below demonstrates that it is possible to apply seminaive evaluation to each of the SCCs of a program's rule-dependence graph individually, rather than to all the rules of the program at once, provided that the order of the SCCs is chosen carefully.

Example 16 Let F be a given database of facts and let P be the program of Example 15. The rule-dependence graph of P (illustrated in Figure 5.4) clearly comprises four SCCs:

$$\begin{aligned} s_1 &= \{r_1, r_2\} \\ s_2 &= \{r_3, r_4\} \\ s_3 &= \{r_5, r_6, r_7\} \\ s_4 &= \{r_8, r_9, r_{10}, r_{11}\} \end{aligned}$$

Now observe that, while each rule in s_4 is clearly dependent on every rule in the program, the rules of $(s_1 \cup s_2 \cup s_3)$ are all *independent* of the rules of s_4. Intuitively, then, it is possible to compute the least Herbrand model of F and P by first computing the least Herbrand model $M_{1,2,3}$ of F and $(s_1 \cup s_2 \cup s_3)$, and then augmenting $M_{1,2,3}$ so that it satisfies all the rules of s_4. In other words, the least Herbrand model of F and P may be computed as the least Herbrand model of $M_{1,2,3}$ and s_4.

A similar argument may be applied, in two different ways, to the computation of $M_{1,2,3}$:

1. Since the rules of s_1 and s_3 are independent of the rules of s_2, $M_{1,2,3}$ may be computed as the least Herbrand model of $M_{1,3}$ and s_2, where $M_{1,3}$ is the least Herbrand model of F and $(s_1 \cup s_3)$.

2. Alternatively, since the rules of s_1 and s_2 are independent of the rules of s_3, $M_{1,2,3}$ may be computed as the least Herbrand model of $M_{1,2}$ and s_3, where $M_{1,2}$ is the least Herbrand model of F and $(s_1 \cup s_2)$.

Finally, note that $M_{1,3}$ may be computed as the least Herbrand model of M_1 and s_3, where M_1 is the least Herbrand model of F and s_1. Similarly, $M_{1,2}$ may be computed as the least Herbrand model of M_1 and s_2.

In summary, therefore, it is possible to compute the least Herbrand model of F and P by first approximating the model as F, and then augmenting the model until it is complete,

by applying seminaive evaluation to the SCCs of P in the order (s_1, s_2, s_3, s_4) or in the order (s_1, s_3, s_2, s_4). □

Note that both of the evaluation strategies presented in Example 16 above succeed in meeting the objectives stated in Section 5.1:

- once seminaive evaluation has been applied to an SCC, the rules of that SCC are not applied again; thus the algorithm avoids repeatedly applying rules when they cannot generate new facts;
- when the rules of any given SCC are applied, *all* the facts which can be used by those rules, with the exception of those generated by the rules themselves, are already present in the database.

Note too that Example 16 requires that the SCCs of the program's rule dependence graph be *ordered* for the purposes of applying seminaive evaluation to the individual SCCs. In fact, the two alternative SCC-orderings suggested in the example are the only SCC-orderings which can guarantee completeness of the evaluation, because they are the only orderings which satisfy the following constraints:

- since both s_2 and s_3 contain rules which are dependent on the rules of s_1, seminaive evaluation must be applied to s_1 before it can be applied to either s_2 or s_3;
- since the rules of s_4 are dependent on the rules of s_1, s_2 and s_3, seminaive evaluation must be applied to all three of these SCCs before it can be applied to s_4.

These observations suggest that, when seminaive evaluation is applied to the individual SCCs of a program's rule-dependence graph, it must be applied to the SCCs in an order which respects the dependencies amongst the program's rules. In particular, let s_1 and s_2 be any two SCCs in a program's rule-dependence graph and assume that s_2 contains a rule which is dependent on some rule in s_1. Then seminaive evaluation must be applied to s_1 before it is applied to s_2. Formally, it is possible to define a binary relation over the set of SCCs in a program's rule-dependence graph to describe the dependencies amongst those SCCs. Since this relation turns out to be a partial order relation, it is possible to apply topological sorting to the set of SCCs to obtain a suitable SCC-ordering.

Definition 18 (SCC-dependence relation) *Let P be a HiLog program with rule-dependence graph G_P and let S_{G_P} denote the set of SCCs in G_P. Then the SCC-dependence relation of S_{G_P}, $\preceq$, is a binary relation over S_{G_P} defined as follows:*

$$\preceq = \{(s_1, s_2) \in S_{G_P} \times S_{G_P} \mid G_P \text{ contains a path from a vertex in } s_1 \text{ to a vertex in } s_2\}$$

□

Example 17 The SCC-dependence relation of the set $\{s_1, s_2, s_3, s_4\}$ of SCCs in the rule-dependence graph of Example 15 is as follows:

$$\preceq = \{(s_1, s_1), (s_1, s_2), (s_1, s_3), (s_1, s_4), (s_2, s_2), (s_2, s_4), (s_3, s_3), (s_3, s_4), (s_4, s_4)\}$$

□

Theorem 18 *Let P be a HiLog program with rule-dependence graph G_P and let S_{G_P} be the set of SCCs in G_P. Then the SCC-dependence relation, $\preceq$, of S_{G_P} is a partial order relation.*

Proof:

$\preceq$ **is reflexive** each SCC of G_P contains at least one vertex, and G_P always contains a path (of length zero) from that vertex to itself;

$\preceq$ **is antisymmetric** assume that S_{G_P} contains two distinct SCCs s_1 and s_2 s.t. $s_1 \preceq s_2$ and $s_2 \preceq s_1$; then, by the definition of $\preceq$, G_P contains a path from a vertex v_1 in s_1 to a vertex v_2 in s_2 and a path from a vertex v_2' in s_2 to a vertex v_1' in s_1; but this implies that G_P contains a path from every vertex in s_1 to every vertex in s_2, and vice-versa, so that s_1 and s_2 cannot be distinct SCCs; it follows that $\preceq$ is antisymmetric;

$\preceq$ **is transitive** assume that $s_1 \preceq s_2$ and $s_2 \preceq s_3$; then, by the definition of $\preceq$, G_P contains a path from a vertex v_1 in s_1 to a vertex v_2 in s_2 and a path from a vertex v_2' in s_2 to a vertex v_3 in s_3; but, since G_P must also contain a path from v_2 to v_2', it follows that G_P contains a path from v_1 to v_3, so that $s_1 \preceq s_3$.

□

Now, given a set of SCCs in a program's rule-dependence graph and an SCC-dependence relation $\preceq$ defined over that set, it is necessary to find an ordering of the SCCs in the set which is consistent with $\preceq$. If it is assumed that each SCC may be identified by means of a natural number subscript, then the ordering may be defined in terms of a permutation function over the set of subscripts.

Definition 19 (Permutation function respecting $\preceq$) *Let S_{G_P} be the set of SCCs in the rule-dependence graph G_P of a HiLog program P; in particular, let $S_{G_P} = \{s_1, \ldots, s_m\}$, where $m \in N$, $m \geq 1$. Now let $\preceq$ be the SCC-dependence relation of S_{G_P}. Then ρ is a "permutation function which respects $\preceq$" if it is a one-to-one mapping with both domain and range equal to $\{k \in N \mid 1 \leq k \leq m\}$ and having the following property: given any $i, j \in N$ s.t. $1 \leq i < j \leq m$, $s_{\rho(j)} \not\preceq s_{\rho(i)}$.* □

Example 18 Let S_{G_P} be the set $\{s_1, s_2, s_3, s_4\}$ of SCCs appearing in the rule-dependence graph of Example 15 and let $\preceq$ be the SCC-dependence relation of S_{G_P}, as shown in Example 17. Then the only two permutation functions which respect $\preceq$ are

- $\{(1,1),(2,2),(3,3),(4,4)\}$ and
- $\{(1,1),(2,3),(3,2),(4,4)\}$.

Note that these functions permute the subscripts of (s_1, s_2, s_3, s_4) to produce the SCC-orderings which were obtained intuitively in Example 16. □

It is now possible to provide an informal overview of an algorithm for evaluating *HiLog* in a manner which takes advantage of rule-dependencies to improve efficiency. This approach to bottom-up evaluation is illustrated for Datalog programs in [34]. Let F be a finite set of *HiLog* facts and let P be a *HiLog* program defined in terms of a finite set of definite *HiLog* Horn clause rules. Then, assuming that it is finite, the least Herbrand model of F and P may be computed using the following procedure:

- construct the rule-dependence graph $G_P = (P, E)$ of P by examining each ordered pair (r_1, r_2) in $P \times P$ and adding it to E if, and only if, the head of r_1 unifies with at least one subgoal in the body of r_2;

- compute the set S_{G_P} of all strongly connected components in G_P—this may be accomplished by means of the algorithm described in [1].
- compute the SCC-dependence relation $\preceq$ of S_{G_P} and use topological sorting to find an SCC-ordering which respects $\preceq$;
- approximate the least Herbrand model of F and P as F;
- complete the model by applying seminaive evaluation to each SCC, in order, ensuring that each SCC sees all the facts already added to the evolving Herbrand model.

The following section describes the final step of the procedure in greater detail and proves that the algorithm is correct and complete.

5.3 Algorithms

This section discusses more formally the process of applying seminaive evaluation to the individual SCCs of a *HiLog* program. It begins by presenting a pseudocode definition of a function which performs the task, and concludes with a number of theorems which prove that the function yields a result which is both correct and complete.

The function sccs_semi, detailed in Figure 5.5, accepts as arguments a finite set F of *HiLog* facts and a *HiLog* program P, defined in terms of a finite set of definite *HiLog* Horn clause rules. It invokes the least_semi function of the previous chapter to apply seminaive evaluation to each SCC of G_P and references the same three global sets of *HiLog* ground terms, I^{old}, I^{Δ} and I^{new}, which are used by least_semi. It does not return a value, but computes the least Herbrand model of F and P into I^{old}.

Theorem 19 *Let L be a language of HiLog with Herbrand Universe H_L. Let $F \in \mathcal{P}(H_L)$, let P be a program in L, comprising only definite HiLog Horn clause rules, and let M be the least Herbrand model of F and P. Now let F' be a subset of F, let P' be a subset of P and let M' be the least Herbrand model of F' and P'. Then $M' \subseteq M$.*

Proof: If t is an element of M', then, by Theorem 3, t is an element of every Herbrand interpretation which is a model of F' and P'. Now, since every Herbrand interpretation

```
1:  void sccs_semi(F, P)
    /* F is a finite set of HiLog ground terms; P is a finite set of
    definite HiLog Horn clause rules; I^old, I^Δ and I^new are global sets of
    HiLog ground terms. */
2:  {
3:      I^new = F;
4:      i = 1;

        /* Let G_P be the rule dependence graph of P; let S_{G_P} be the set
        of all strongly-connected components in G_P; specifically, let
        S_{G_P} be the set {s_1, ..., s_m}; let ≼ be the SCC-dependence
        relation for the SCC's of G_P and let ρ be a permutation function,
        defined over the subscripts of {s_1, ..., s_m}, which respects ≼. */

5:      while (TRUE)
6:      {
7:          least_semi(s_{ρ(i)});
8:          if (i == m)
9:              break;
10:         I^new = I^old;
11:         i++;
12:     }
13: }
```

Figure 5.5: Procedure for SCC-based seminaive evaluation

which is a model of F and P is clearly also a model of F' and P', it follows that t is an element of every such interpretation. Hence, by Theorem 3, t is an element of M. □

Theorem 20 *Let L be a HiLog language with Herbrand Universe H_L. Let $F \in \mathcal{P}(H_L)$ and let P be a HiLog program in L comprising only definite HiLog Horn clause rules. Let M be the least Herbrand model of F and P. Then, after execution of* sccs_semi(F, P), $I^{old} \subseteq M$.

Proof: Assume that the execution of the while-loop of lines 5–12 results in sequence of calls to least_semi with arguments $s_1, \ldots, s_m$. For each $i \in N$, $1 \leq i \leq m$, let I_i^{old} denote the value of I^{old} after the execution of least_semi(s_i). Then it is easily proved by induction that each I_i^{old} is a subset of M.

For the basis, note that when least_semi(s_1) is invoked, $I^{new} = F$, owing to the assignment of line 3. It follows from Theorem 14 that I_1^{old} is a subset of the least Herbrand model M_1 of F and s_1. But, since $F \subseteq M$ and s_1 is a subset of P, M_1 is a subset of M by Theorem 19. Thus I_1^{old} is a subset of M.

For the inductive step, assume that, for some $i \in N$, $1 \leq i < m$, $I_i^{old} \subseteq M$. Then, when least_semi(s_{i+1}) is invoked, $I^{new} = I_i^{old}$, owing to the assignment of line 10. It follows from Theorem 14 that I_{i+1}^{old} is a subset of the least Herbrand model M_{i+1} of I_i^{old} and s_{i+1}. But, since I_i^{old} is a subset of M, by the inductive hypothesis, and since s_{i+1} is a subset of P, M_{i+1} is a subset of M by Theorem 19. Thus I_{i+1}^{old} is a subset of M.

When sccs_semi terminates, I^{old} is clearly equal to I_m^{old}, so the theorem is proved. □

Theorem 21 *Let L be a HiLog language with Herbrand Universe H_L. Let $F \in \mathcal{P}(H_L)$ and let P be a HiLog program in L comprising only definite HiLog Horn clause rules. Let M be the least Herbrand model of F and P and assume that M is finite. Then* sccs_semi(F, P) *will terminate with $I^{old} = M$.*

Proof: Let S_{G_P} be the set of all strongly-connected components in the rule-dependence graph of P. Specifically, let $S_{G_P} = \{s_1, \ldots, s_m\}$, where $m \in N$, $m \geq 1$. Let $\preceq$ denote the SCC-dependence relation defined over S_{G_P} and let ρ be a permutation function, defined

over the subscripts of $\{s_1, \ldots, s_m\}$, which respects $\preceq$. For each $i \in N$, $1 \leq i \leq m$, let I_i^{old} be the value of I^{old} after the call to least_semi($s_{\rho(i)}$) on line 7 of sccs_semi. It is readily proved by induction that each such I_i^{old} is a model of F and $\bigcup_{j=1}^{i} s_{\rho(j)}$.

For the basis, observe that, owing to the assignment on line 3, $I^{new} = F$ when least_semi($s_{\rho(1)}$) is invoked. Thus it follows from Theorem 16 that I_1^{old} is a model of F and $s_{\rho(1)}$. Since $\bigcup_{j=1}^{1} s_{\rho(j)}$ is just $s_{\rho(1)}$, this proves the basis.

For the inductive step, assume that, for some $n \in N$, $1 \leq n < m$, I_n^{old} is a model of F and $\bigcup_{j=1}^{n} s_{\rho(j)}$. Observe that $I_n^{old} \subseteq M$, as demonstrated in Theorem 20, and that $s_{\rho(n+1)} \subseteq P$, so that, by Theorem 19, the least Herbrand model M' of I_n^{old} and $s_{\rho(n+1)}$ is a subset of M and is thus finite. Now the assignment on line 10 ensures that, when least_semi($s_{\rho(n+1)}$) is invoked, $I^{new} = I_n^{old}$, so that, by Theorem 16, least_semi($s_{\rho(n+1)}$) will duly terminate with $I^{old} = M'$. Since I_{n+1}^{old} is, by definition, equal to the value of I^{old} when least_semi($s_{\rho(n+1)}$) terminates, $I_{n+1}^{old} = M'$ and thus clearly satisfies all the rules in $s_{\rho(n+1)}$. Now assume that I_{n+1}^{old} does not satisfy $\bigcup_{j=1}^{n} s_{\rho(j)}$. Note that I_{n+1}^{old} is a superset of I_n^{old} and, by the inductive hypothesis, I_n^{old} satisfies every rule in $\bigcup_{j=1}^{n} s_{\rho(j)}$. So, if I_{n+1}^{old} fails to satisfy $\bigcup_{j=1}^{n} s_{\rho(j)}$, it must be because $(I_{n+1}^{old} - I_n^{old})$ contains terms which enable I_{n+1}^{old} to satisfy the body of some rule in $\bigcup_{j=1}^{n} s_{\rho(j)}$ without satisfying the head of that rule. Observe, however, that every ground term in $(I_{n+1}^{old} - I_n^{old})$ is an instance of the head of some rule in $s_{\rho(n+1)}$, so the rule-dependence graph of P must contain an edge from some rule in $s_{\rho(n+1)}$ to some rule in $\bigcup_{j=1}^{n} s_{\rho(j)}$. This, in turn, implies that, for some $j \in N$, $1 \leq j \leq n$, $s_{\rho(n+1)} \preceq s_{\rho(j)}$, contradicting the assertion that ρ respects $\preceq$. The contradiction forces the conclusion that I_{n+1}^{old} satisfies $\bigcup_{j=1}^{n} s_{\rho(j)}$, as well as $s_{\rho(n+1)}$, and so the inductive step is proved.

Since $I^{old} = I_m^{old}$ when sccs_semi terminates, and since $\bigcup_{j=1}^{m} s_{\rho(j)} = P$, I^{old} must be a model of F and P when sccs_semi terminates. Therefore, by the definition of "least model", $M \subseteq I^{old}$. Furthermore, $I^{old} \subseteq M$ by Theorem 20, so $I^{old} = M$ and the proof is complete. □

5.4 The sccs System

The sccs system is a modified version of the semi system described in Section 4.3 and implements the SCC-based seminaive evaluation algorithm described in this chapter.

5.4.1 System Organization

This is as for the semi system, except that the evaluation component has been modified slightly so that it applies seminaive evaluation to individual SCCs, in an order respecting the SCC-dependency relation, rather than to the entire program. Furthermore, the system includes the following additional components:

Unify component This is a C implementation of an algorithm for testing the unifiability of two *HiLog* terms. The algorithm is similar to that described in [21] in the context of unification of FOL atoms.

Graph component It is implemented in C and responsible for:

- testing the unifiability of rule heads and rule subgoals and constructing a rule-dependence graph accordingly;
- identifying SCCs in the rule-dependence graph (using an algorithm described in [1]);
- computing the program's SCC-dependence relation (Definition 18) and
- using topological sorting to find an ordering of SCCs which respects the SCC-dependence relation.

5.4.2 Database Usage

This is as for the semi system (Section 4.3).

5.5 SCC-based Evaluation vs Simple Seminaive Evaluation

The objectives of the SCC-based evaluation algorithm are stated in terms of avoiding unnecessary rule applications, so it seems reasonable to use "number of rule applications" as a means of quantifying the performance gains which the algorithm enjoys over the simple seminaive evaluation algorithm of the previous chapter. It is also worth noting that, in a *HiLog* evaluation system implemented on a relational database platform, a rule application may involve the execution of several database queries, including at least one

join-based query, so that a reduction in the number of rule applications can have significant practical implications.

This section begins by describing a procedure for calculating the maximum number of rule applications required by the *simple* seminaive evaluation algorithm to compute the model of a *HiLog* program, given the details of the SCC-based evaluation of the program and, in particular, the number of iterations required by the evaluation of each SCC. The section concludes by using this procedure to deduce that, while the performance gains obtained by using the SCC-based algorithm can be quite modest for some categories of programs, they can be very substantial for others.

5.5.1 Analysing the Worst-case behaviour of Simple Seminaive Evaluation

Given the rule dependence graph of a *HiLog* program and the number of iterations of seminaive evaluation applied to each SCC of the program by the SCC-based evaluation algorithm, it is possible to deduce the largest number of iterations required by the simple seminaive evaluation algorithm to compute the model of the program:

1. Construct a "condensed rule-dependence graph" in which each vertex v denotes an SCC of the program, labelled with the number i_v of iterations of seminaive evaluation applied to the SCC by SCC-based evaluation. Include the edge (v_1, v_2) in the graph if, and only if, the SCC denoted by v_2 includes a rule which is directly dependent on a rule in the SCC denoted by v_1.

   Example 19 When SCC-based evaluation is applied to the program of Figure 5.6 (see Figure 5.7 for the rule-dependence graph) and in the presence of an initial database of facts comprising only $a_1(a, b, c, d, e)$, the numbers of iterations of seminaive evaluation applied to each SCC are as denoted by the vertex labels of the condensed rule-dependence graph of Figure 5.8. These values were reported by execution of the sccs system with the program of Figure 5.6 as input. □

2. Convert the condensed rule-dependence graph into a tree by systematically replicating vertices with multiple parents. Figure 5.9 shows the graph obtained by applying

$$
\begin{array}{lrcl}
r_1\text{:} & a_2(X_1,X_2,X_3,X_4,X_5) & :- & a_1(X_1,X_2,X_3,X_4,X_5) \\
r_2\text{:} & a_3(X_1,X_2,X_3,X_4,X_5) & :- & a_2(X_1,X_2,X_3,X_4,X_5) \\
r_3\text{:} & a_4(X_1,X_2,X_3,X_4,X_5) & :- & a_3(X_1,X_2,X_3,X_4,X_5) \\
r_4\text{:} & a_1(X_5,X_1,X_2,X_3,X_4) & :- & a_4(X_1,X_2,X_3,X_4,X_5) \\
 & & & \\
r_5\text{:} & b_1(X_1,X_2,X_3,X_4) & :- & a_4(X_1,X_2,X_3,X_4,a) \\
 & & & \\
r_6\text{:} & b_2(X_1,X_2,X_3,X_4) & :- & b_1(X_1,X_2,X_3,X_4) \\
r_7\text{:} & b_3(X_1,X_2,X_3,X_4) & :- & b_2(X_1,X_2,X_3,X_4) \\
r_8\text{:} & b_4(X_1,X_2,X_3,X_4) & :- & b_3(X_1,X_2,X_3,X_4) \\
r_9\text{:} & b_1(X_4,X_1,X_2,X_3) & :- & b_4(X_1,X_2,X_3,X_4) \\
 & & & \\
r_{10}\text{:} & c_1(X_1,X_2,X_3) & :- & a_4(X_1,X_2,X_3,X_4,a) \\
 & & & \\
r_{11}\text{:} & c_2(X_1,X_2,X_3) & :- & c_1(X_1,X_2,X_3) \\
r_{12}\text{:} & c_3(X_1,X_2,X_3) & :- & c_2(X_1,X_2,X_3) \\
r_{13}\text{:} & c_4(X_1,X_2,X_3) & :- & c_3(X_1,X_2,X_3) \\
r_{14}\text{:} & c_1(X_3,X_1,X_2) & :- & c_4(X_1,X_2,X_3) \\
 & & & \\
r_{15}\text{:} & d_1(X_1,X_2,X_3) & :- & b_4(X_1,X_2,X_3,b), c_4(X_1,X_2,b) \\
 & & & \\
r_{16}\text{:} & d_2(X_1,X_2,X_3) & :- & d_1(X_1,X_2,X_3) \\
r_{17}\text{:} & d_3(X_1,X_2,X_3) & :- & d_2(X_1,X_2,X_3) \\
r_{18}\text{:} & d_4(X_1,X_2,X_3) & :- & d_3(X_1,X_2,X_3) \\
r_{19}\text{:} & d_1(X_3,X_1,X_2) & :- & d_4(X_1,X_2,X_3)
\end{array}
$$

Figure 5.6: Program Illustrating Worst-case Behaviour of Simple Seminaive Evaluation

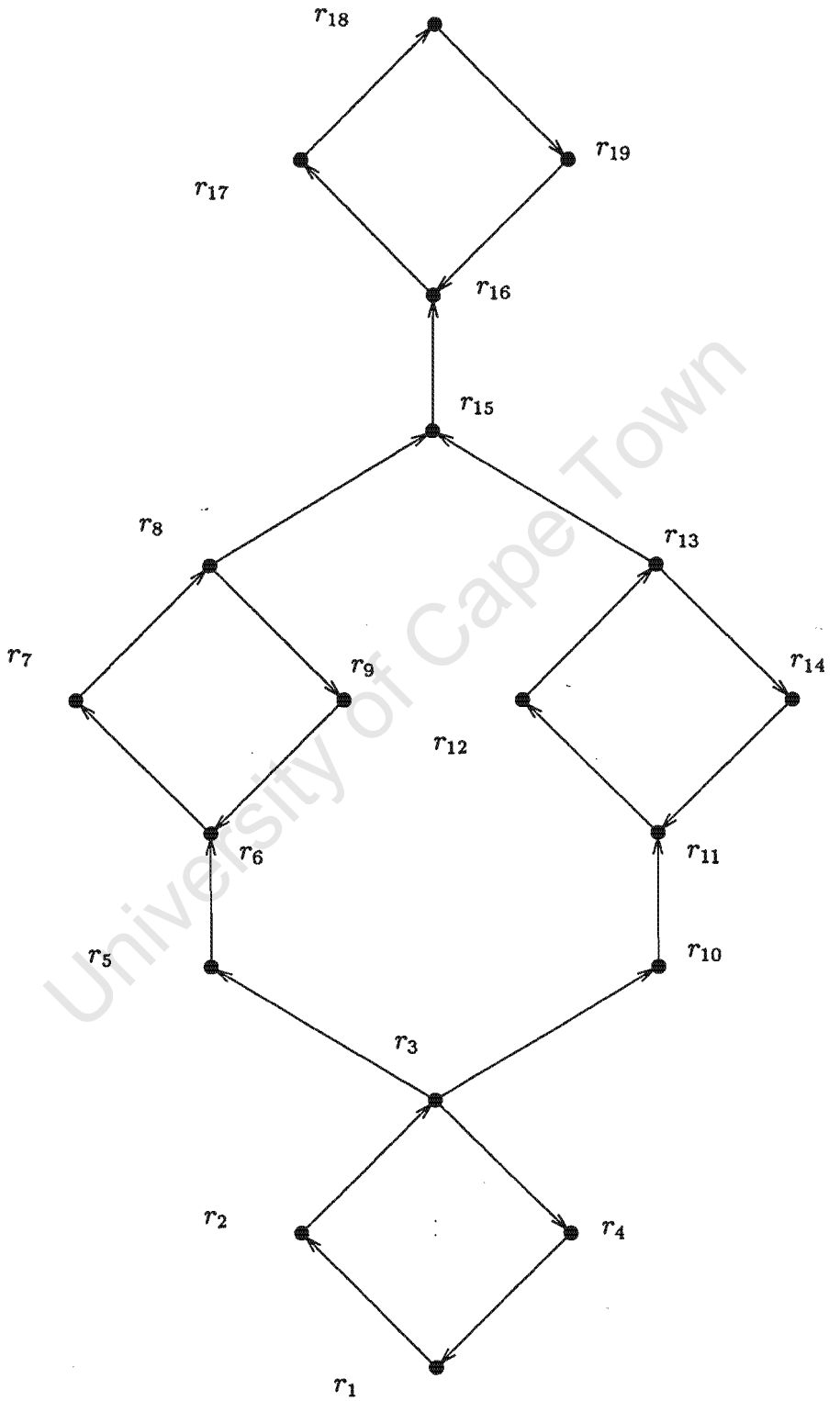


Figure 5.7: Rule Dependence Graph for Program of Figure 5.6

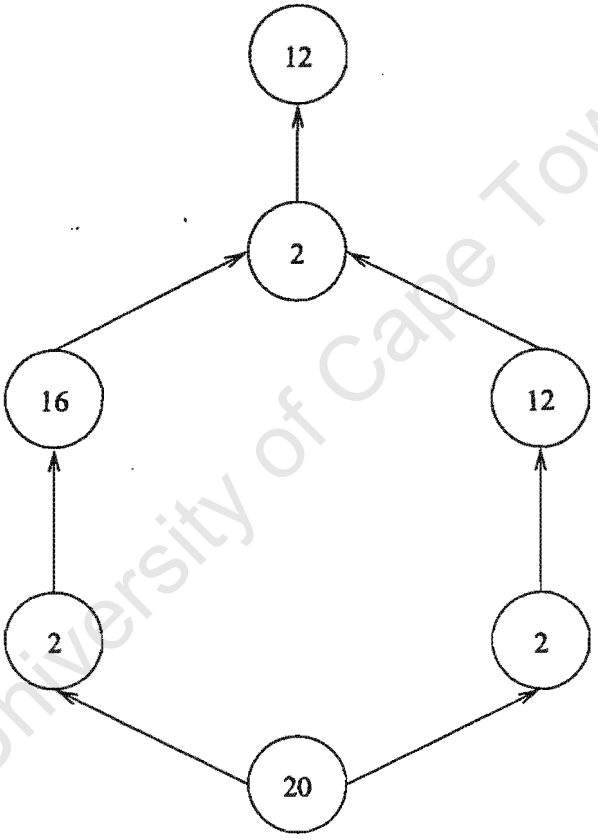


Figure 5.8: Condensed Form of the Rule Dependence Graph of Figure 5.7

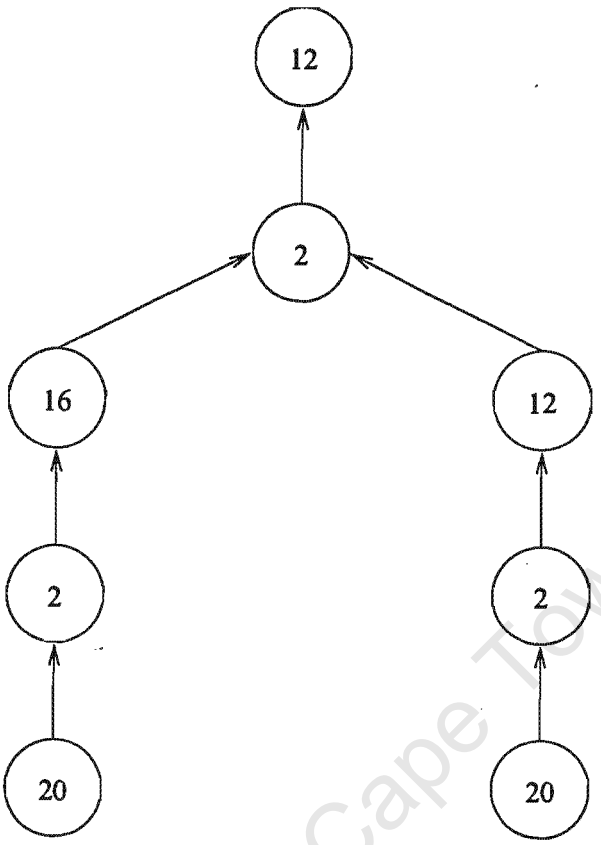


Figure 5.9: Tree Form Condensed Rule Dependence Graph derived from the Graph of Figure 5.8

this procedure to the DAG of Figure 5.8.

3. Apply the following formula to establish the number of iterations required by simple seminaive evaluation to compute the model of the program denoted by the tree rooted at vertex v:

$$I(v) = \begin{cases} i_v + \max(I(v_1), \ldots, I(v_k)) - 1 & \text{if } v \text{ is a nonleaf vertex with children} \\ & v_1, \ldots, v_k; \\ i_v & \text{if } v \text{ is a leaf vertex} \end{cases}$$

The rationale behind this formula is that, in the worst case, the simple seminaive evaluation algorithm will only be able to begin producing facts using the rules of some SCC v on the iteration following the *last* iteration on which one of v's children yields a new fact. This case is illustrated by the program of Figure 5.6: note, for example, that the application of rules r_5 and r_{10} can only yield new facts when the

model contains a fact with a_4 as the functor and a as the fifth argument. Such a fact is produced only on iteration 19, the last iteration on which a rule of the SCC $\{r_1, r_2, r_3, r_4\}$ yields a new fact.

Example 20 The total number of iterations required by simple seminaive evaluation to compute the least Herbrand model of $a_1(a, b, c, d, e)$ and the program of Figure 5.6 is $I(v_{root})$, where v_{root} is the root of the tree-form condensed rule-dependence graph of Figure 5.9:

$$\begin{aligned} I(v_{root}) &= 12 + (2 + \max((16 + 2 + 20 - 1 - 1), (12 + 2 + 20 - 1 - 1)) - 1) - 1 \\ &= 12 + 2 + (16 + 2 + 20 - 1 - 1) - 1 - 1 \\ &= 12 + 2 + 36 - 1 - 1 \\ &= 48 \end{aligned}$$

This number agrees with the total number of iterations required by simple seminaive evaluation to complete the computation, as reported by an execution of the semi system with the program of Figure 5.6 as input. □

It is very easy to see that the worst-case behaviour of the simple seminaive evaluation algorithm will be at its worst if the program's rule-dependence graph comprises more than one SCC and the SCC-dependence relation is a total order: on any given iteration the rules of only one SCC yield new facts, even though all the program rules are applied. Under these circumstances, the number of iterations of simple seminaive evaluation required is clearly $(\sum_{j=1}^{n} i_j) - (n - 1)$, where the $i_1, \ldots, i_n$ are the iterations required by the SCC-based evaluation algorithm for each of the program's n SCCs. Provided that the evaluation of at least one of the program's SCCs requires a large number of iterations, the $(n - 1)$ term will be small in comparison with the remainder of the expression and may safely be discarded. Then the total number of *rule applications*, RA_{semi}, required by simple seminaive evaluation may be estimated as $\sum_{j=1}^{n} i_j \sum_{j=1}^{n} r_j$, where the $r_1, \ldots, r_n$ denote the number of rules in each of the program's SCCs. The number of rule applications, RA_{scc}, required by SCC-based evaluation, on the other hand, is just $\sum_{j=1}^{n} i_j r_j$.

Clearly, RA_{scc} can never exceed RA_{semi}, while the difference between RA_{semi} and RA_{scc} can be either negligible or very substantial, depending on the nature of the program:

- If the program comprises only one SCC, the SCC-based algorithm is equivalent to the simple seminaive algorithm and $\mathrm{RA}_{semi} = \mathrm{RA}_{scc}$.

- If the program comprises two SCCs and one of these has only one rule and requires only one iteration, while the other has a large number r of rules and requires a large number i of iterations, then $\mathrm{RA}_{semi} = ir + i + r + 1$ and $\mathrm{RA}_{scc} = ir + 1$. Here the difference between RA_{semi} and RA_{scc}, $(i + r)$, is small when compared with the number of rule applications required by either algorithm.

- Assume that the program comprises n SCCs, where $n \in N$, $n > 1$, and that $i_1, \ldots, i_n$ are the numbers of iterations applied to the SCCs by the SCC-based algorithm. Assume, furthermore, that the SCCs all have a comparable number of rules, so that it is reasonable to use a constant r to denote the number of rules in each SCC. Then

$$\begin{aligned} \mathrm{RA}_{semi} &= nr(i_1 + \cdots + i_n) \\ \mathrm{RA}_{scc} &= r(i_1 + \cdots + i_n) \end{aligned}$$

  so that

$$\frac{\mathrm{RA}_{semi}}{\mathrm{RA}_{scc}} = n$$

  In other words, the number of rule applications required by simple seminaive evaluation exceeds that required by the SCC-based algorithm by a factor equal to the number of SCCs in the program.

- Consider a program comprising $2n$ SCCs, $s_1, \ldots, s_{2n}$, where $n \in N$ and $n \geq 1$. Now assume that SCCs with odd-numbered subscripts each require some large number a of iterations but each have only one rule, while SCCs with even-numbered subscripts each require only one iteration but each have a rules. Then

$$\begin{aligned} \mathrm{RA}_{semi} &= n(a+1)n(a+1) \\ &= n^2(a+1)^2 \end{aligned}$$

  and

$$\mathrm{RA}_{scc} = 2na$$

These circumstances are admittedly rather contrived, but they serve to illustrate that the number of rule applications required by the simple seminaive evaluation algorithm can exceed that required by the SCC-based algorithm by an order of magnitude.

In conclusion, the SCC-based algorithm is superior to the simple seminaive evaluation algorithm in that, while its application will never require more rule applications than simple seminaive evaluation, it may require substantially fewer rule applications.

Chapter 6

General Seminaive Evaluation

The SCC-by-SCC seminaive evaluation procedure of the previous chapter represented an improvement upon simple seminaive evaluation because it could substantially reduce the number of rule applications needed to compute the least model of a given set of facts and rules. It accomplished this by analysing the rule dependencies in the program and applying iterative evaluation to each maximal set of mutually recursive rules, rather than to the entire program. This chapter presents an evaluation procedure which also endeavours to reduce the number of rule applications, but which focuses on the iterative evaluation applied to each SCC in a program's rule-dependence graph. The algorithm is essentially analogous to the GSN evaluation algorithm described for Datalog evaluation in [30].

Section 6.1 examines the drawbacks of the conventional seminaive evaluation procedure and states the objectives of the procedure described in this chapter. Section 6.2 develops an intuitive description of the procedure and Section 6.3 defines the underlying algorithms of the procedure formally and proves the procedure correct and complete. Section 6.4 describes the gsn system, an enhanced version of the sccs system (Section 5.4) based on the GSN algorithm. The chapter concludes with an investigation of the impact of rule-orderings on the performance of the GSN algorithm based on data from gsn. The relative efficiencies of simple seminaive and GSN evaluation are compared by means of theoretical analysis.

6.1 Motivation and Objectives

Refer once again to Examples 8 and 9 of Chapter 4, which detail the application of naive evaluation and seminaive evaluation, respectively, to a simple program describing paths in a directed graph in terms of the graph's edge set.

$$\begin{aligned} r_1\colon\ & p(X,Y) \ :-\ e(X,Y) \\ r_2\colon\ & p(X,Z) \ :-\ e(X,Y), p(Y,Z) \end{aligned}$$

Observe that, on the first iteration of naive evaluation, the first rule is applied to the facts denoting the graph edges to generate all those facts denoting paths of length one. Since these facts are added to the evolving model immediately, the procedure can, in the same iteration, apply the second rule to both the "edge facts" and the "path facts" to generate facts denoting paths of length two.

By contrast, the application of the second rule on the first iteration of *seminaive* evaluation cannot generate any facts, because the rule application examines only I^{old} and I^{Δ}, while the "path facts" generated by applying the first rule remain in I^{new} until they are transferred to I^{Δ} at the beginning of the second iteration. Only on this iteration can they be used, by the application of the second rule, to generate the facts denoting paths of length two.

The following example illustrates more dramatically the disadvantages of withholding newly-generated facts from the evolving model until the beginning of the next iteration.

Example 21 Let F be a set comprising the single *HiLog* fact $p(a,b)(c)$ let P be the following set of definite *HiLog* Horn clause rules:

$$\begin{aligned} r_1\colon\ & p2(X,Y)(Z) \ :-\ p1(X,Y)(Z) \\ r_2\colon\ & p3(X,Y)(Z) \ :-\ p2(X,Y)(Z) \\ r_3\colon\ & p4(X,Y)(Z) \ :-\ p3(X,Y)(Z) \\ r_4\colon\ & p5(X,Y)(Z) \ :-\ p4(X,Y)(Z) \\ r_5\colon\ & p1(Y,X)(Z) \ :-\ p5(X,Y)(Z) \end{aligned}$$

Now consider the computation of the least Herbrand model of F and P by means of the naive and seminaive evaluation procedures.

Assuming that, on each iteration, naive evaluation applies the rules in the order shown, the computation proceeds as follows:

> on the first iteration, the application of r_1 yields the fact $p2(a,b)(c)$; since this fact is added to the evolving model immediately, it is available to the application of r_2 on the first iteration, which uses it to produce $p3(a,b)(c)$; this fact, in turn, is used by r_3 to produce $p4(a,b)(c)$, and so on, until the application of r_5 at the end of the first iteration yields $p1(b,a)(c)$; evaluation proceeds in a similar fashion on the second iteration, and, since the application of r_5 on this iteration yields only the facts $p1(a,b)(c)$ and $p1(b,a)(c)$, which are already in the model, no new facts are generated on the third and last iteration.

Now consider the manner in which the seminaive evaluation procedure computes the model:

> at the beginning of the first iteration, $(I^{old} \cup I^{\Delta})$ contains only the base fact, $p1(a,b)(c)$, which the first application of r_1 uses to generate $p2(a;b)(c)$; however, since this new fact is placed in I^{new}, where it is inaccessible to the seminaive rule-application procedure, the first application of r_2 cannot use it to generate any new facts; indeed, $(I^{old} \cup I^{\Delta})$ remains equal to $\{p1(a,b)(c)\}$ for the entire duration of the first iteration, and so it is clear that r_1 is the only rule whose application can generate a new fact on this iteration; at the beginning of the second iteration $p2(a,b)(c)$ is transferred to I^{Δ}, where it can be used by the second application of r_2 to generate $p3(a,b)(c)$; once again, however, the new fact remains unavailable to the rule-application procedure until the next iteration and so no other rule applications generate any new facts on the second iteration; it is easy to see, by extrapolation, that the fifth iteration is the first on which the application of r_5 produces a new fact, $p1(b,a)(c)$, and that a further four iterations are needed to generate all the new facts which may be derived from this one; only by the end of the ninth iteration has the complete model been computed, and only after the tenth iteration, which yields no new facts, does the evaluation terminate.

□

Example 21 above demonstrates that, while the seminaive evaluation algorithm of Chapter 4 is able to ensure the non-repetition property by maintaining a partitioning of the evolving model into "new" and "old" facts, it does so at the expense of being able to make newly-generated facts *immediately* available for use in the generation of further facts, with the result that the algorithm can require many more iterations to compute a model than naive evaluation would need to compute the same model. This chapter describes an evaluation algorithm which manages to perform as well as naive evaluation, in terms of the number of iterations required to compute a model, without sacrificing the non-repetition property of conventional seminaive evaluation. The algorithm is based on the General Seminaive (GSN) algorithm for Datalog evaluation described in [30] and will be referred to by the same name in this work.

6.2 Overview

This section provides an informal overview of GSN evaluation, a procedure which successfully reconciles immediate updates of the evolving Herbrand model with the non-repetition property. It demonstrates why a straightforward modification of the seminaive rule-application procedure cannot suffice and then describes a way of solving the problem by employing an alternative scheme for representing the model and the program rules. The section concludes by showing how GSN evaluation uses the new representations to compute the least Herbrand model of a given set of facts and rules.

It is tempting to suppose that the objectives of GSN evaluation may be met by simply modifying the seminaive rule application procedure so that, instead of adding newly-generated facts to I^{new}, it adds them directly to I^{Δ}, provided they are not already in $(I^{old} \cup I^{\Delta})$. Certainly, if it were true at the beginning of an iteration that I^{old} contained only facts seen by *every* rule, while I^{Δ} contained only facts seen by *no* rule, then each rule application performed on that iteration would avoid making any derivations which violated the non-repetition property. However, consider the situation which would prevail once all the rules had been applied: all the facts in I^{Δ}, except those which had been in the

set at the beginning of the iteration, would be *new* with respect to the first rule; on the other hand, all the facts in I^{Δ}, except those which had been generated by applying the last rule, would be *old* with respect to the last rule. So it would be impossible to transfer facts from I^{Δ} to I^{old} in preparation for the next iteration.

This illustrates that, if an evaluation procedure adds newly-generated facts to an evolving model immediately, it becomes impossible to partition the model into a set of facts which are "old" with respect to *every* rule and a set of facts which are "new" with respect to *every* rule, because, in general, the partitioning must be effected differently for each rule. To solve the problem, it is necessary to find a way of determining, for any given rule, which facts it has "seen" and which it has not.

The GSN evaluation algorithm accomplishes this by associating with each ground term added to the evolving model a "time-stamp" which records when the ground term was added to the model. A time-stamp is also associated with each rule of the program in order to record when the rule was last applied. Thus, given a program rule and a ground term in the model, it is possible to determine whether or not the rule has "seen" the ground term by simply comparing the time-stamps of the rule and the ground term.

More specifically, let F be a finite set of facts in a *HiLog* language L, let P be a *HiLog* program in L, defined in terms of a finite set of definite *HiLog* Horn clause rules, and assume that it is necessary to compute the least Herbrand model of F and P using GSN evaluation. The algorithm represents the model by means of a global relation I over a scheme comprising two attributes: the first attribute, denoted by \$1, has as its domain the Herbrand universe H_L of L; the second attribute, denoted by \$2, has as its domain the set of all nonnegative integers. Thus each fact in the evolving Herbrand model is represented by an ordered pair $(t, stamp)$, where t is the fact itself and *stamp* is its associated time-stamp value. The algorithm also uses an integer variable to record the time-stamp value of each rule of the program and maintains a global integer variable, *count*, to keep track of the "passage of time" and a global Boolean variable, *new_terms*, to record the addition of new ground terms to the model.

The steps involved in the evaluation are as follows:

- construct the rule-dependence graph G_P of P as described in the previous chapter;

- compute the set S_{G_P} of all strongly connected components in G_P;
- compute the SCC-dependence relation of S_{G_P}, as defined in the previous chapter, and use topological sorting to find an SCC-ordering which respects the relation;
- place each of the ground terms of F in the global I relation along with a time-stamp value of 0;
- initialise the global *count* variable to 0;
- apply GSN evaluation to each SCC in S_{G_P}, in order, as follows:
  - set the time-stamp value of each rule in the SCC to 0; this ensures that all the facts currently present in the evolving model will initially by seen as "new" by each rule in the SCC;
  - apply each rule in the SCC to the model and, immediately after applying each rule, set its time-stamp value equal to *count* to ensure that all the facts which the rule-application procedure used are duly regarded by subsequent applications of the rule as having been "seen"; the SCC-evaluation procedure relies on the rule-application procedure to use a rule's time-stamp value, in conjunction with those of the model's facts, to distinguish between the facts which the rule has already seen and those which it has not; it also assumes that, immediately prior to adding new facts to the model, the rule-application procedure will increment *count* and use the variable's new value as the time-stamp value of each new fact, thus recording the fact as more recent than any of those used by the rule-application; finally, it assumes that, if any new facts are added to the model, the procedure will indicate this by setting the global variable *new_terms* to TRUE;
  - if new facts were added to the model, return to the previous step.

The following section describes the algorithms underlying GSN evaluation more formally and proves that the procedure is correct and complete.

6.3 Algorithms

6.3.1 The GSN Rule Application Algorithm

The purpose of the GSN rule application algorithm is to apply a given definite *HiLog* Horn clause rule to an evolving Herbrand model and to add the newly-generated facts to the model so that they are immediately available to subsequent rule applications. To ensure that GSN evaluation exhibits the non-repetition property, the algorithm must be able to distinguish between those facts which the rule has already seen and those which it has not and must avoid making any derivations based exclusively on the seen facts.

Figure 6.1 details the algorithm by means of the pseudocode function gsn_apply which accepts as arguments c, a definite *HiLog* Horn clause rule, and *rule_stamp*, the rule's associated time-stamp value. The function operates in the presence of the global variables I, *count* and *new_terms*, as described in the previous section, and ν, the global variable used by the term-matching function of Chapter 3 to represent variable assignments. It does not return a value, but affects the global state by adding newly-generated facts to I, incrementing *count* to record the passage of time and setting *new_terms* to TRUE if it adds any terms to I.

Note that the operation of gsn_apply is very similar to that of semi_apply, the function for conventional seminaive rule application described in Chapter 4, in that it too comprises three steps which closely mirror those of semi_apply:

1. for each subgoal A_i of the rule, use term-matching to compute three relations, r_i^{old}, r_i^{Δ} and r_i^{full}, whose tuples denote variable assignments under which I satisfies A_i; r_i^{old} is based exclusively on facts already seen by the rule; r_i^{Δ} is based on facts which the rule has not previously seen; r_i^{full} is the union of r_i^{old} and r_i^{Δ}; the relations are created and computed by lines 3 to 18;

2. join the tuples of the relations computed in step 1 to obtain tuples representing variable assignments under which I satisfies the entire rule body, avoiding the computation of any tuples based solely on tuples from the r_i^{old} relations (lines 22 and 23);

```
1:  void gsn_apply(c, rule_stamp)
    /* Apply a rule to the evolving Herbrand model; c is a definite
    HiLog Horn clause rule; rule_stamp is an integer used to partition
    the terms of I into "old facts" and "new facts"; I is a global set
    of tuples (t, stamp), where t is a ground term and stamp is an
    integer; count is a global integer variable; ν is the global
    variable assignment accessed by match; new_terms is a global
    Boolean variable. */
2:  {
        /* Let c be the clause A_0 ∨ ¬A_1 ∨ ... ∨ ¬A_n, where n ∈ N, n ≥ 1
        and A_0, ..., A_n are nonground HiLog atomic formulas. */
3:      for (i = 1; i <= n; i++)
            /* Let the set of distinct variable symbols in A_i be
            {v_{i_1}, ..., v_{i_{m_i}}}. */
4:          create empty relations r_i^old, r_i^Δ and r_i^full over the
5:          relation scheme (v_{i_1}, ..., v_{i_{m_i}});

        /* Let I be the relation {(t_1, stamp_1), ..., (t_f, stamp_f)}. */
6:      for (j = 1; j <= f; j++)
7:          for (k = 1; k <= n; k++)
8:          {
9:              ν = ∅;
10:             if (match(A_k, t_j))
11:             {
12:                 r_k^full = r_k^full ∪ {τ_ν};
13:                 if (stamp_j >= rule_stamp)
14:                     r_k^Δ = r_k^Δ ∪ {τ_ν};
15:                 else
16:                     r_k^old = r_k^old ∪ {τ_ν};
17:             }
18:         }

19:     count++;

20:     for (p = 1; p <= n; p++)
21:     {
22:         create a relation r_body = π_{v_1,...,v_q}(r_1^old ⋈ ... r_{p-1}^old ⋈ r_p^Δ ⋈ r_{p+1}^full ⋈ ... ⋈ r_n^full),
23:         where {v_1, ..., v_q} is the set of distinct variables in A_0;

            /* Let r_body be the set {u_1, ..., u_w}. */
24:         for (h = 1; h <= w; h++)
25:             if (A_0ψ_{u_h} ∉ π_{$1}(I))
26:             {
27:                 I = I ∪ {(A_0ψ_{u_h}, count)};
28:                 new_terms = TRUE;
29:             }
30:     }
31: }
```


Figure 6.1: Procedure for applying a rule under GSN evaluation

3. use each tuple computed in step 2 to substitute for the variables of the rule head, adding the derived fact to I if it is not already represented in I (lines 24 to 29).

However, observe that, while semi_apply could rely on the set, either I^{old} or I^{Δ}, which contained a fact to identify the fact as "seen" or "unseen", gsn_apply must make the distinction by comparing the fact's time-stamp value to the time-stamp value indicating when the rule was last applied (conditional statement of lines 13 to 16). Also, while semi_apply simply added derived facts to I^{new}, gsn_apply must add the facts to I, along with a time-stamp value based on *count* (line 27), and, prior to adding any facts to I, it must increment *count* (line 19) to ensure that all the new facts are recorded as "more recent" than any of the facts used by the rule application. Finally, note that gsn_apply is required to indicate the derivation of any new facts by setting *new_terms* to TRUE (line 28).

Theorem 22 below proves a constraint on the set of facts which an invocation of gsn_apply can add to I.

Theorem 22 *Assume that the function* gsn_apply *is invoked in the presence of the global relation I and with c and rule_stamp as arguments, where c is a HiLog Horn clause rule and rule_stamp is a natural number. Let $I^{old} = \pi_{\$1}(\sigma_{\$2<rule\_stamp}(I))$ and let $I^{\Delta} = \pi_{\$1}(\sigma_{\$2\geq rule\_stamp}(I))$. Let J be the set of all new tuples added to I by the invokation of* gsn_apply. *Then $\pi_{\$1}(J) \subseteq T_c(\pi_{\$1}(I))$ and, furthermore, $\pi_{\$1}(I \cup J) \supseteq (T_c(I^{old} \cup I^{\Delta}) - T_c(I^{old}))$.*

Proof: Let c be a definite *HiLog* Horn clause in a *HiLog* language L, where L has Herbrand Universe H_L. Specifically, let $c = A_0 \vee \neg A_1 \vee \ldots \vee \neg A_n$, where $n \in N$, $n \geq 1$ and $A_0, \ldots, A_n$ are nonground *HiLog* terms. Let the set of distinct variable symbols in A_0 be $\{v_1, \ldots, v_q\}$ and, for each $i \in N$, $1 \leq i \leq n$, let the set of distinct variable symbols in A_i be $\{v_{i_1}, \ldots, v_{i_{m_i}}\}$. Now let $\Lambda_{c,I^{\rm old},I^{\Delta}} = \bigcup_{i=1}^{n}\{A_0\mu \mid \tau_\mu \in \pi_{v_1,\ldots,v_q}(s_1^{old} \bowtie \ldots \bowtie s_{i-1}^{old} \bowtie s_i^{\Delta} \bowtie s_{i+1}^{full} \bowtie \ldots \bowtie s_n^{full})\}$, where, for each $i \in N$, $1 \leq i \leq n$, s_i^{old}, s_i^{Δ} and s_i^{full} are defined as follows: each is a relation over the relation scheme $(v_{i_1}, \ldots, v_{i_{m_i}})$ in which each attribute has H_L as its domain; specifically:

- $s_i^{old} = \{u_i \in {H_L}^{m_i} \mid A_i\psi_{u_i} \in I^{old}\} = \{\tau_\nu \in {H_L}^{m_i} \mid A_i\nu \in I^{old}\}$

- $s_i^{\Delta} = \{u_i \in {H_L}^{m_i} \mid A_i\psi_{u_i} \in I^{\Delta}\} = \{\tau_\nu \in {H_L}^{m_i} \mid A_i\nu \in I^{\Delta}\}$
- $s_i^{full} = s_i^{old} \cup s_i^{\Delta}$

Now Theorem 12 proves that $T_c(I^{old} \cup I^{\Delta}) - T_c(I^{old}) \subseteq \Lambda_{c,I^{\text{old}},I^{\Delta}} \subseteq T_c(I^{old} \cup I^{\Delta}) = T_c(I)$. Also, every tuple added to I by gsn_apply is added by the for-loop of lines 24–29, and the for-loop adds to I a tuple representing each $A_0\psi_{u_h}$ which is computed on line 25 and which is not already represented in I. Clearly, then, it is possible to complete the proof of the theorem by showing that the set of all $A_0\psi_{u_h}$ computed on line 25 is equal to $\Lambda_{c,I^{\text{old}},I^{\Delta}}$.

Observe that $\Lambda_{c,I^{\text{old}},I^{\Delta}}$ may be rewritten as $\bigcup_{p=1}^{n}\{A_0\psi_{u_h} \mid u_h \in \pi_{v_1,\ldots,v_q}(s_1^{old} \bowtie \ldots \bowtie s_{p-1}^{old} \bowtie s_p^{\Delta} \bowtie s_{p+1}^{full} \bowtie \ldots \bowtie s_n^{full})\}$. Furthermore, for each $p \in N$, $1 \leq p \leq n$, the for-loop of lines 24–29 computes, on line 25, each $A_0\psi_{u_h}$ s.t. $u_h \in r_{body}$. Thus it suffices to show that, for each $p \in N$, $1 \leq p \leq n$, the r_{body} relation computed on line 22 is equal to $\pi_{v_1,\ldots,v_q}(s_1^{old} \bowtie \ldots \bowtie s_{p-1}^{old} \bowtie s_p^{\Delta} \bowtie s_{p+1}^{full} \bowtie \ldots \bowtie s_n^{full})$. This is clearly true if, for each $i \in N$, $1 \leq i \leq n$, $s_i^{old} = r_i^{old}$, $s_i^{\Delta} = r_i^{\Delta}$ and $s_i^{full} = r_i^{full}$.

Note that the for-loop of lines 3–5 creates, for each $i \in N$, $1 \leq i \leq n$, relations r_i^{old}, r_i^{Δ} and r_i^{full} over the same relation scheme over which s_i^{old}, s_i^{Δ} and s_i^{full} are defined.

Now, in the nested for-loop of lines 6–18, the compound statement of lines 8–18 is executed for each ground term, t_j, in $\pi_{\$1}(I)$ and each subgoal, A_k, in the rule body. It follows from the correctness and completeness of the match function (Theorems 4 and 5) that the compound statement of lines 11–17 is executed if, and only if, $A_k\nu = t_j$. Then, if $t_j \in I^{\Delta}$, $stamp_j \geq rule\_stamp$ (by the definition of I^{Δ}) and so the condition on line 13 tests true and line 14 places τ_ν in r_k^{Δ}. Otherwise $t_j \in I^{old}$ and line 16 places τ_ν in r_k^{old}. Also, the assignment on line 12 ensures that every τ_ν which is placed in either r_k^{old} or r_k^{Δ} is also placed in r_k^{full}. So, it is not difficult to see that, after the execution of the nested for-loop, the following equalities hold for each $k \in N$, $1 \leq k \leq n$:

- $r_k^{old} = \{\tau_\nu \in {H_L}^{m_i} \mid A_k\nu \in I^{old}\} = s_k^{old}$
- $r_k^{\Delta} = \{\tau_\nu \in {H_L}^{m_i} \mid A_k\nu \in I^{\Delta}\} = s_k^{\Delta}$
- $r_k^{full} = r_k^{old} \cup r_k^{\Delta} = s_k^{old} \cup s_k^{\Delta} = s_k^{full}$

This completes the proof of the theorem. □

```
1:  void eval_scc(S)
    /* Apply the rules of an SCC to the evolving Herbrand model. S is
    a set of HiLog rules denoting a strongly-connected component of a
    program's rule-dependence graph; new_terms is a global Boolean
    variable which may be set TRUE by gsn_apply; count is a global
    integer variable which may be incremented by gsn_apply. */
2:  {
        /* Let S be the set {c_1, ..., c_n}. */

3:      for (i = 1; i <= n; i++)
4:          last_stamp_i = 0;

5:      do
6:      {
7:          new_terms = FALSE;

8:          for (j = 1; j <= n; j++)
9:          {
10:             gsn_apply(c_j, last_stamp_j);
11:             last_stamp_j = count;
12:         }
13:     }
14:     while (new_terms);
15: }
```

Figure 6.2: Procedure for evaluating an SCC under GSN evaluation

6.3.2 GSN Evaluation of an SCC

Application of GSN evlauation to each SCC of a *HiLog* program's rule graph is effected by calling the eval_scc function of Figure 6.2 for each SCC. The function accepts as an argument the set S of definite *HiLog* Horn clause rules which constitutes the SCC and operates in the presence of the global variables I, *count* and *new_terms*, as defined in Section 6.2. It does not return a value, but adds elements to I so that $\pi_{\$1}(I)$ satisfies S.

Note that eval_scc bears comparison to the least_semi function (of Chapter 4) which is applied to an entire program in the simple seminaive evaluation procedure of Chapter 4 and to each SCC of a program's rule-dependence graph in the SCC-based seminaive evaluation procedure of Chapter 5.

However, while least_semi ages facts for all rules at once by transferring facts from I^{Δ} to

I^{old} at the end of each iteration, eval_scc ages facts on a per-rule basis by updating the time-stamp value associated with each rule. Inspection of gsn_apply shows that, immediately after a call to the function on line 10 of eval_scc, the value of *count* is equal to the time-stamp value assigned to each of the facts *generated* by the rule application and greater than that assigned to any fact *used* by the rule application. Since a fact is deemed to be unseen by a rule only if its time-stamp value is greater than or equal to the rule's time-stamp value (line 13 of gsn_apply), the assignment on line 11 of eval_scc ensures that all the facts which have been used by the rule application are duly aged, while those generated by the rule application are still regarded as "unseen" by the rule.

The function terminates at the end of an iteration if no call to gsn_apply succeeds in setting *new_terms* to TRUE. Theorems 23, 24 and 25 below prove that, if eval_scc is invoked with $\pi_{\$1}(I) = F$, the function terminates with $\pi_{\$1}(I)$ equal to the least model of F and S, provided that the least model is finite.

Theorem 23 *Assume that the function* eval_scc *is invoked in the presence of the global variable I and with argument s, where s is a finite, nonempty set of definite HiLog Horn clause rules. Let I_1 be the value of I immediately prior to the invokation of* eval_scc *and let $I_1' = \pi_{\$1}(I_1)$. Now let M be the least Herbrand model of I_1' and s. Then, throughout the execution of* eval_scc, *$\pi_{\$1}(I)$ remains a subset of M.*

Proof: The proof is a straightforward induction on the number of calls to gsn_apply on line 10 of eval_scc.

First observe that, prior to the first call to gsn_apply, $\pi_{\$1}(I) = I_1'$ and, since I_1' is necessarily a subset of M, $\pi_{\$1}(I) \subseteq M$. This completes the proof of the basis.

For the inductive step, assume that $\pi_{\$1}(I)$ is a subset of M after k calls to gsn_apply and let J be the set of tuples added to I by the $(k+1)$st call to gsn_apply. By Theorem 22, $\pi_{\$1}(J)$ is a subset of $T_c(\pi_{\$1}(I))$ and, by Theorem 8 and the inductive hypothesis, $T_c(\pi_{\$1}(I))$ is a subset of M. Clearly, then, the value of $\pi_{\$1}(I)$ remains a subset of M after the $(k+1)$st call to gsn_apply and so the inductive step is proved. □

Theorem 24 *Assume that the function* eval_scc *is executed in the presence of the global variable I. Now consider any call to* gsn_apply$(c_j, last_stamp_j)$ *on line 10 of the function*

and let the value of I immediately prior to the call be I_j. Let $I_j^{old} = \pi_{\$1}(\sigma_{\$2<last\_stamp_j}(I_j))$ and let $I_j^{\Delta} = \pi_{\$1}(\sigma_{\$2\geq last\_stamp_j}(I_j))$. Then $T_{c_j}(I_j^{old}) \subseteq \pi_{\$1}(I_j)$.

Proof: The proof is an induction on the number of the iteration of the do-loop of lines 5–14 on which the call to gsn_apply is executed.

For the inductive step, assume that the theorem holds on every iteration of the do-loop with a number less than or equal to k. Now consider the execution of gsn_apply(c_j, $last\_stamp_j$) on iteration $k+1$ of the do-loop and assume that the ground term t is an element of $T_{c_j}(I_j^{old})$. If $c_j = A_0 \vee \neg A_1 \vee \ldots \vee \neg A_m$, where $m \in N$, $m \geq 1$ and $A_0, \ldots, A_m$ are nonground *HiLog* terms, then, by the definition of T_{c_j}, there exists a variable assignment ν under which $t = A_0\nu$ and $A_1\nu, \ldots, A_m\nu$ are all elements of I_j^{old}. Observe that, in accordance with the definition of I_j^{old}, each of the $A_1\nu, \ldots, A_m\nu$ must have an associated "stamp value" which is less than $last\_stamp_j$. It follows that each of the $A_1\nu, \ldots, A_m\nu$ must have been an element of $\pi_{\$1}(I_j)$ on iteration k of the do-loop, so that t must have been an element of $T_{c_j}(I_j^{old} \cup I_j^{\Delta})$ when gsn_apply(c_j, $last\_stamp_j$) was executed on iteration k of the do-loop. Either t was then an element of $T_{c_j}(I_j^{old})$, in which case it was already an element of $\pi_{\$1}(I_j)$, by the inductive hypothesis, or it was an element of $T_{c_j}(I_j^{old} \cup I_j^{\Delta}) - T_{c_j}(I_j^{old})$, in which case it was an element of $\pi_{\$1}(I)$ immediately after the call to gsn_apply, by Theorem 22. Either way, t is clearly in $\pi_{\$1}(I_j)$ on iteration $k+1$ of the do-loop, and so the inductive step is proved.

For the basis, note that, on the first iteration of the do-loop, $last\_stamp_j = 0$ for every call to gsn_apply. Since every ground term represented in I has a "stamp value" greater than or equal to zero, it follows from the definition of I_j^{old} that $I_j^{old} = \emptyset$, in which case $T_{c_j}(I_j^{old}) = \emptyset$ and the theorem is trivially satisfied.

This completes the proof of the theorem. □

Theorem 25 *Assume that the function* eval_scc *is invoked in the presence of the global variable I and with argument s, where s is a finite, nonempty set of definite HiLog Horn clause rules. Let I_1 be the value of I immediately prior to the invokation of* eval_scc *and let $I_1' = \pi_{\$1}(I_1)$. Now let M be the least Herbrand model of I_1' and s. Then, if M is finite, the execution of* eval_scc *terminates with $\pi_{\$1}(I) = M$.*

Proof: Observe that the assignment on line 7 of eval_scc sets the value of the global variable *new_terms* to FALSE at the beginning of each iteration of the do-loop of lines 5–14. Furthermore, the condition on line 25 of gsn_apply and the assignment on line 28 of gsn_apply ensure that the value of *new_terms* is set to TRUE if, and only if, a rule application adds tuples representing new *HiLog* ground terms to I. However, Theorem 23 proves that $\pi_{\$1}(I)$ remains a subset of M, so that, if M is finite, new tuples cannot be added to I indefinitely—eventually the condition on line 14 will test false and the algorithm will terminate.

It can be proved, by contradiction, that, when the execution of eval_scc terminates, $\pi_{\$1}(I)$ is a model of I_1' and s. First assume that this is not the case. Since tuples are never deleted from I, $\pi_{\$1}(I)$ must be a superset of I_1', so that, if $\pi_{\$1}(I)$ is not a model of I_1' and s, it must be because it fails to satisfy some rule c_j in s. Let $c_j = A_0 \vee \neg A_1 \vee \ldots \vee \neg A_n$, where $n \in N$, $n \geq 1$ and $A_0, \ldots, A_n$ are nonground *HiLog* terms. Then there exists a variable assignment ν s.t. $A_1\nu, \ldots, A_n\nu$ are all elements of $\pi_{\$1}(I)$, but $A_0\nu$ is not an element of $\pi_{\$1}(I)$. Note that no new tuples could have been added to I on the last iteration of the do-loop of lines 5–14, since otherwise the condition on line 14 would not have tested false at the end of the iteration. It follows that, when gsn_apply(c_j, $last\_stamp_j$) was executed on the last iteration, $A_1\nu, \ldots, A_n\nu$ were all elements of $\pi_{\$1}(I)$. Now let I_j denote the value of I immediately prior to the call to gsn_apply(c_j, $last\_stamp_j$). Let $I_j^{old} = \pi_{\$1}(\sigma_{\$2 < last\_stamp_j}(I_j))$ and let $I_j^{\Delta} = \pi_{\$1}(\sigma_{\$2 \geq last\_stamp_j}(I_j))$. Then $A_1\nu, \ldots, A_n\nu$ are clearly elements of $(I_j^{old} \cup I_j^{\Delta})$, so that $A_0\nu$ must be an element of $T_{c_j}(I_j^{old} \cup I_j^{\Delta})$. Furthermore, $A_0\nu$ cannot be an element of $T_{c_j}(I_j^{old})$, since that would imply, by Theorem 24, that $A_0\nu$ were an element of $\pi_{\$1}(I_j)$, contradicting the assumption that $\pi_{\$1}(I)$ does not satisfy c_j. Thus $A_0\nu$ must have been an element of $T_{c_j}(I_j^{old} \cup I_j^{\Delta}) - T_{c_j}(I_j^{old})$ when gsn_apply(c_j, $last\_stamp_j$) was executed on the last iteration of the do-loop. Then, by Theorem 22, a tuple containing $A_0\nu$ must have been added to I by the execution of gsn_apply(c_j, $last\_stamp_j$)—this again contradicts the assumption that $\pi_{\$1}(I)$ does not satisfy c_j and forces the conclusion that, when eval_scc terminates, $\pi_{\$1}(I)$ is indeed a model of I_1' and s.

Finally, note that, according to the definition of "least model", $M \subseteq \pi_{\$1}(I)$. Also, by Theorem 23, $\pi_{\$1}(I) \subseteq M$, so $\pi_{\$1}(I) = M$ when execution of eval_scc terminates and the theorem is proved. □

```
1:  void sccs_gsn(F, P)
    /* Computes the least Herbrand model of F and P by means of
    SCC-by-SCC GSN evaluation. F is a finite set of HiLog ground
    terms; P is a finite set of definite HiLog Horn clause rules;
    I is a global relation comprising tuples of the form (t, stamp),
    where t is a ground term and stamp is an integer; count is a
    global integer variable. */
2:  {
3:      I = {(t, 0) | t ∈ F};
4:      count = 0;

        /* Let G_P be the rule dependence graph of P; let S_{G_P} be the
        set of all strongly-connected components in G_P; specifically,
        let S_{G_P} be the set {s_1, ..., s_m}; let ≺ be the SCC-dependence
        relation for the SCC's of G_P and let ρ be a permutation
        function, defined over the subscripts of {s_1, ..., s_m},
        which respects ≺. */

5:      for (i = 1; i <= m; i++)
6:          eval_scc(s_{ρ(i)});
7:  }
```

Figure 6.3: Procedure for SCC-by-SCC GSN evaluation

6.3.3 GSN Evaluation of a Program

The sccs_gsn function of Figure 6.3 accepts as arguments a finite set F of *HiLog* facts and a finite set P of definite *HiLog* Horn clause rules and operates in the presence of the global variables I and *count*, as defined in Section 6.2. The function does not return a value, but updates I so that $\pi_{\$1}(I)$ is equal to the least Herbrand model of F and P.

Lines 3 and 4 initialise the global variables, adding each fact in F to I, along with a time-stamp value of 0, and setting the "timer variable" *count*to 0. The remainder of the algorithm simply invokes eval_scc to apply GSN evaluation to each SCC in P's rule-dependence graph, in an order that respects the SCC-dependence relation.

Note that, while the sccs_semi function of Chapter 5 must ensure that I^{new} contains all the facts in the evolving model prior to the evaluation of each SCC (line 10 of sccs_semi), so that all these facts are initially regarded as unseen by each rule of the SCC, GSN evaluation can simply rely on lines 3 and 4 of eval_scc, which set the time-stamp value of

each rule to 0, to render each fact in the model unseen by any rule in the SCC.

Theorems 26 and 27 prove that, if the least Herbrand model M of F and P is finite, sccs_gsn terminates with $\pi_{\$1}(I)$ equal to M.

Theorem 26 *Let L be a HiLog language with Herbrand Universe H_L. Let $F \in \mathcal{P}(H_L)$ and let P be a HiLog program in L comprising only definite HiLog Horn clause rules. Let M be the least Herbrand model of F and P. Then, if* sccs_gsn *is invoked in the presence of the global variable I and with F and P as arguments, $\pi_{\$1}(I)$ remains a subset of M.*

Proof: This follows easily from Theorems 19 and 23. Details of the proof are omitted since it is very similar to the proof of Theorem 20. □

Theorem 27 *Let L be a HiLog language with Herbrand Universe H_L. Let $F \in \mathcal{P}(H_L)$ and let P be a HiLog program in L comprising only definite HiLog Horn clause rules. Let M be the least Herbrand model of F and P and assume that M is finite. Then, if* sccs_gsn *is invoked in the presence of the global variable I and with F and P as arguments,* sccs_gsn *will terminate with $\pi_{\$1}(I) = M$.*

Proof: This follows from Theorems 19, 25 and 26. Details of the proof are omitted since it is virtually identical to the proof of Theorem 21. □

6.4 The gsn System

The gsn system is a modified version of the sccs system described in Section 5.4 and implements the GSN evaluation algorithm described in this chapter.

6.4.1 System Organization

This is identical to that of the sccs system (Section 5.4), except that the evaluation component is based on the gsn_apply and eval_scc procedures (Figures 6.1 and 6.2 respectively).

6.4.2 Database Usage

This is identical to that of the sccs system, except that, while sccs requires three single-column tables, corresponding to the I^{old}, I^{Δ} and I^{new} relations, to store various subsets of the evolving model, gsn requires only one two-column table. The first column stores string values denoting *HiLog* ground terms and the second stores integers denoting "timestamp values" assigned to the ground terms as required by gsn_apply (Figure 6.1).

6.5 Performance Analysis: GSN vs simple seminaive evaluation

The GSN evaluation algorithm aims to improve the efficiency of *HiLog* program evaluation by reducing the number of iterations which must be applied to each of a program's SCCs. Thus it seems reasonable to assess the performance of the algorithm by comparing the number of iterations required by GSN evaluation with the number required by simple seminaive evaluation when each algorithm is applied to a program comprising a single SCC. In [30] it is noted that

- reducing the number of iterations also reduces the number of rule applications which, in turn, reduces the overhead of database access and join computation and
- reducing the number of iterations required by an evaluation without altering the number of derivations it performs increases its "set-orientedness" and reduces the number of I/O operations.

This section begins by arguing that, unless it applies the rules of an SCC in an appropriate order, the GSN evaluation algorithm will not prove significantly more efficient than simple seminaive evaluation. The section then cites a formal analysis of the impact of rule-ordering on evaluation efficiency ([30]) and describes a practical experiment, conducted with the gsn system, which yields results which are in accordance with the formal analysis. It concludes by demonstrating that the number of iterations required by simple seminaive evaluation to compute the closure of an SCC can exceed that required by GSN evaluation by a factor comparable to the number of rules in the SCC.

Example 21 in this chapter demonstrated that the naive evaluation algorithm of Chapter 3 could evaluate the program of that example using substantially fewer iterations than required by the seminaive evaluation algorithm of Chapter 4, provided that each iteration applied the rules in the order shown. Inspection of the rules readily reveals that if the order in which they are applied on each iteration is reversed, only facts produced by r_5 can be used (by r_1) in the same iteration. Most of the potential benefits of GSN evaluation are lost and the algorithm will perform little better than simple seminaive evaluation.

The authors of [30] identify a class of rule orderings, the "fair, static orderings", in the context of which it is possible to conduct a rigorous formal analysis of the influence of rule orderings on the efficiency of GSN evaluation. Definitions and results in [30] which are relevant to this section's performance analyses are reproduced below.

Definition 20 *If an SCC S comprises the rules $R_1, \ldots, R_n$, a fair static ordering is an ordering $(R_{i_1}, \ldots, R_{i_n})$ where $i_1, \ldots, i_n$ is a permutation of $1, \ldots, n$.* □

Definition 21 *If C is a cycle within an SCC and O is a fair ordering of the rules of the SCC, O is said to break C by degree $B(C,O) = i$ if i is the least number such that, for some cyclic permutation O_1 of O, C is a subsequence of O_1^i. If an ordering breaks a cycle by degree 1, it is said to* preserve the cycle. □

Definition 22 *Let $\lhd$ be a relation defined on the class of fair orderings so that, given any two fair orderings O_1 and O_2 on a rule-dependence graph G, $O_1 \lhd O_2$ if, for every simple cycle C in G, $B(C,O_1) \leq B(C,O_2)$.* □

Definition 23 *If G is a rule-dependence graph and O_1 and O_2 are static, fair orderings of the rules of G such that $O_1 \lhd O_2$,*

$$MaxR(O_1, O_2, G) = \max\{B(C,O_2)/B(C,O_1) \mid C \text{ is a simple cycle in } G\}$$

□

Theorem 28 *Given an SCC S, any two fair orderings O_1 and O_2, such that $O_1 \lhd O_2$, and any set of base facts, let the number of iterations required to compute the closure of S by*

$$
\begin{array}{lllll}
r_1\text{:} & p_2(X_1,X_2,X_3,X_4,X_5) & :- & p_1(X_1,X_2,X_3,X_4,X_5) \\
r_2\text{:} & p_3(X_1,X_2,X_3,X_4,X_5) & :- & p_2(X_1,X_2,X_3,X_4,X_5) \\
r_3\text{:} & p_4(X_1,X_2,X_3,X_4,X_5) & :- & p_3(X_1,X_2,X_3,X_4,X_5) \\
r_4\text{:} & p_1(X_5,X_1,X_2,X_3,X_4) & :- & p_4(X_1,X_2,X_3,X_4,X_5), p_5(X_1,X_2,X_3,X_4,X_5) \\
r_5\text{:} & p_5(X_1,X_2,X_3,X_4,X_5) & :- & p_2(X_1,X_2,X_3,X_4,X_5)
\end{array}
$$

Figure 6.4: Single-SCC Program for Investigating Rule Orderings

bottom-up fixpoint evaluations using rule orderings O_1 and O_2 be n_1 and n_2 respectively. n_1 and n_2 are related as $n_1 - k \leq n_2 \leq MaxR(O_1, O_2, G) \cdot n_1 + k$, where k is bounded by the length of the longest acyclic path in the rule graph for the SCC.

Proof: See [30]. □

It follows from this theorem that if O denotes an ordering of the rules of G which is minimal under $\lhd$, then it is possible to observe a roughly linear relationship between $MaxR(O, O', G)$ and the number of iterations required by O', where O' is any ordering of the rules of G such that $O \lhd O'$. Figure 6.5 reproduces the rule-dependence graph of the single-SCC program of Figure 6.4. Note that the SCC comprises two simple cycles $C_1 = (r_1, r_2, r_3, r_4)$ and $C_2 = (r_1, r_5, r_4)$ and that $O = (r_1, r_2, r_3, r_5, r_4)$ is a fair ordering which preserves both the cycles and which is thus minimal under $\lhd$. Table 6.1 describes evaluations of the program by the gsn system, detailing each evaluation in terms of the rule ordering O' applied (where $O \lhd O'$), $B(C_1, O')$, $B(C_2, O')$, $MaxR(O, O', G)$ and the number of iterations $i_{O'}$ required for the evaluation (minus one, to exclude the last iteration since it does not generate any new facts). The values are as reported by the gsn system. The plot of $i_{O'}$ vs $MaxR(O, O', G)$ in Figure 6.6 clearly illustrates that the number of iterations required can be linearly dependent on $MaxR(O, O', G)$.

6.5.1 Comparison with Seminaive Evaluation

Consider the application of simple seminaive and GSN evaluation to a single SCC of a rule-dependence graph. It is easy to see, and straightforward to prove, that, after any

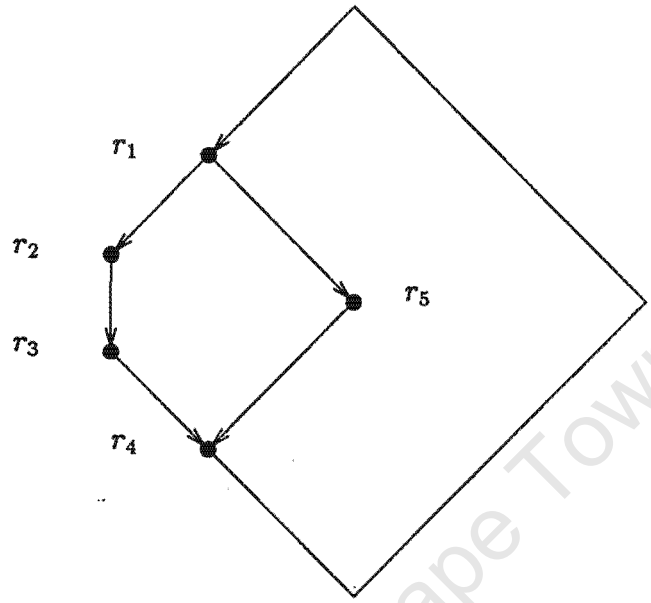


Figure 6.5: Rule-Dependence Graph for Program of Figure 6.4

O'	$B(C_1, O')$	$B(C_2, O')$	$MaxR(O, O', G)$	$i_{O'}$
$(r_1, r_2, r_3, r_5, r_4)$	1	1	1	5
$(r_1, r_2, r_3, r_4, r_5)$	1	2	2	9
$(r_1, r_3, r_2, r_5, r_4)$	2	1	2	10
$(r_3, r_2, r_4, r_5, r_1)$	2	2	2	11
$(r_2, r_1, r_5, r_4, r_3)$	3	1	3	14
$(r_4, r_5, r_3, r_2, r_1)$	3	2	3	15

Table 6.1: Evaluations of the Program of Figure 6.4 with Different Rule Orderings

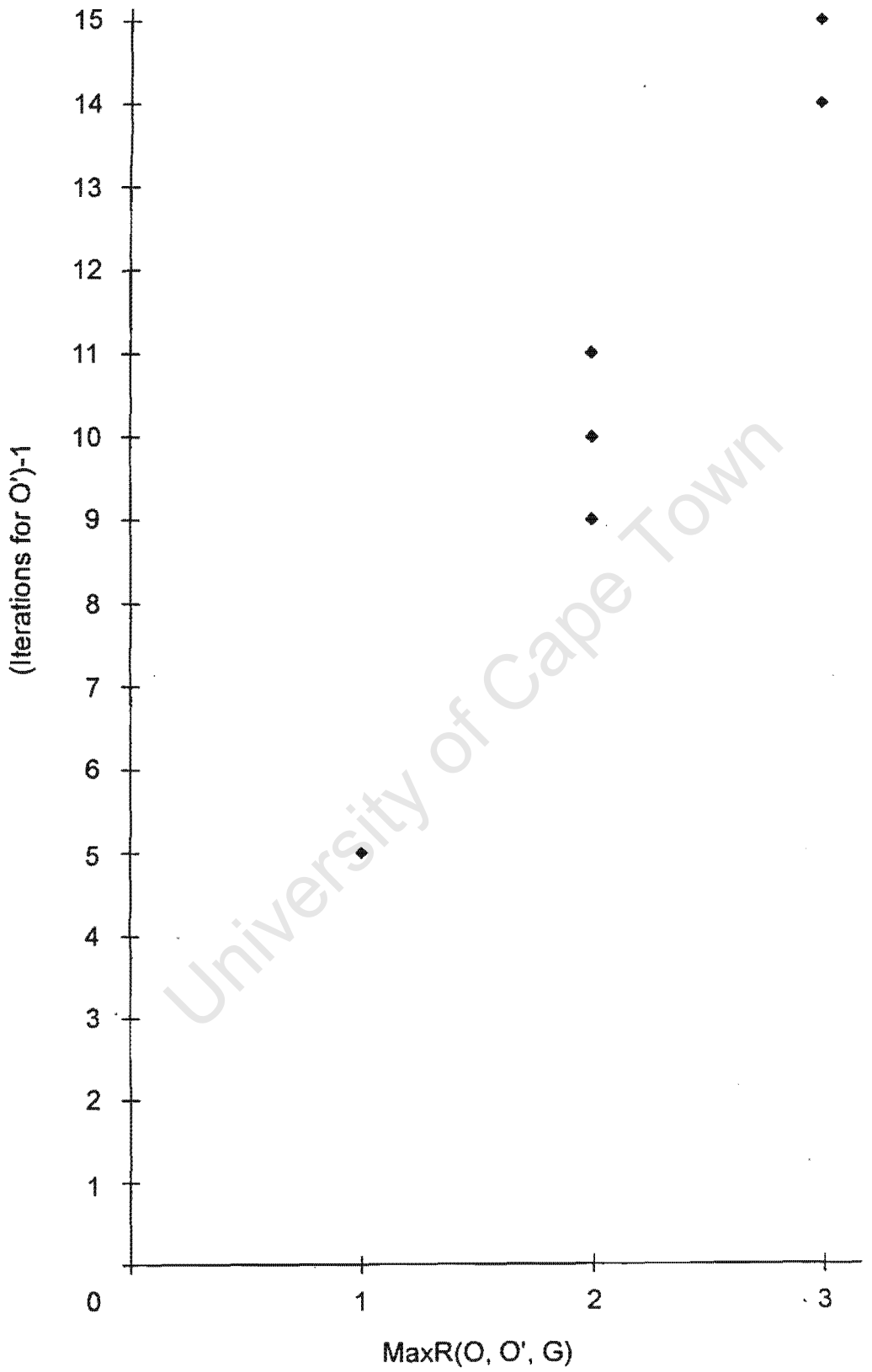


Figure 6.6: (Iterations for O') -1 vs $MaxR(O,O',G)$

number of iterations, the set of facts generated by the GSN algorithm will be a (not necessarily proper) superset of the set of facts generated by simple seminaive evaluation. Therefore the GSN algorithm cannot fare worse than the simple seminaive algorithm in terms of iterations required.

However, the GSN algorithm can complete the evaluation using significantly fewer iterations, especially if it consistently uses a rule-ordering which preserves all the SCC's cycles. Assume that the SCC comprises a single cycle $(r_1, \ldots, r_n)$, that the initial set of facts matches only subgoals of r_1 and that GSN evaluation uses the rule-ordering $(r_1, \ldots, r_n)$ on each of i iterations to complete the evaluation. Since no new facts are generated on the last iteration, the largest number of rule applications which *do* generate new facts is $n(i-1)$. Now the simple seminaive evaluation algorithm will clearly perform the same sequence of derivations, but on each iteration it will use only one rule to generate new facts, so the total number of iterations it requires, including the last iteration (on which no new facts are generated), is $n(i-1)+1$, or $ni-(n-1)$. So it may be contended that, when the number of iterations required by GSN evaluation is sufficiently large, the number of iterations for simple seminaive evaluation can exceed that for GSN evaluation by a factor comparable to the number of rules in the SCC.

Chapter 7

Conclusion

This thesis has presented and examined, both from the perspective of completeness and correctness and from the perspective of efficiency, four *HiLog* evaluation algorithms, each of which is a close analog of a corresponding Datalog evaluation algorithm. In each case, a practical system was developed to verify the feasibility of implementing the algorithm.

The results of these investigations suggest that the algorithms for bottom-up evaluation of Datalog can readily be adapted to the evaluation of *HiLog*. Indeed, *HiLog*'s first order semantics seem to ensure that any Datalog evaluation algorithm which assumes a Herbrand semantics and which is capable of dealing with structured terms will have an analogous *HiLog* evaluation algorithm, provided that Herbrand semantics are also assumed for *HiLog*.

Specifically, the thesis has described and evaluated:

- an algorithm for naive evaluation of *HiLog* programs;
- an algorithm for simple seminaive evaluation of *HiLog* programs; here, performance analyses led to the conclusion that it is possible for the derivations performed by naive evaluation to exceed in number those performed by seminaive evaluation by a *factor* proportional to the number of iterations of seminaive evaluation required;
- an algorithm for SCC-based evaluation of *HiLog*; performance comparisons were based on the number of rule applications performed by SCC-based and simple seminaive evaluation and led to the conclusion that the number of applications required

by the simple algorithm can be an order of magnitude greater than that required by the SCC-based algorithm;

- an algorithm for general seminaive (GSN) evaluation of *HiLog*; performance analyses showed that, when GSN and simple seminaive evaluation are applied to a set of mutually recursive rules, the number of iterations needed by simple seminaive evaluation can exceed the number required by GSN by a factor comparable to the number of rules.

7.1 Further Work

This work has considered only the evaluation of *HiLog* programs restricted to facts and Horn clause rules. An investigation of the evaluation of *HiLog* programs whose rules can incorporate negative body literals might begin with an attempt to adapt the algorithms for evaluating similar Datalog programs. The reader is referred to [34] for a discussion of such algorithms and to [31] for a consideration of issues pertaining to negation in *HiLog*.

The work has also been confined to a consideration of semantic structures equivalent to Herbrand Interpretations. Future work could well investigate the bottom-up evaluation of *HiLog* programs whose semantics are decribed by arbitrary semantic structures, particularly those which support the semantics of equality as defined in [13].

In all the evaluation algorithms presented in this work, the term-matching algorithm considers each fact in the set against which it is applied. This may be contrasted with the equivalent fact-matching step in Datalog evaluation, where knowledge of the predicate symbol in the subgoal for which the matching is performed, and the partitioning of model into multiple relations according to facts' predicates, can limit the number of facts which need to be considered. (Refer to [34] for a description of Datalog rule application.) Future work might consider the implementation of indexing schemes capable of organizing *HiLog* facts on the basis of their structures, as well as on the basis of the constant symbols they contain.

It may also prove worthwhile to investigate the possible existence of rule-rewriting optimizations specifically suited to *HiLog* programs. Since the "Magic Templates" optimiza-

tion described in [29] is capable of taking advantage of the *structures* of terms within Datalog facts to improve evaluation efficiency, and structure information can at times be the only useful information available about a desired set of *HiLog* facts, Magic Templates may serve as an ideal starting point for such research.

Bibliography

[1] A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Company, 1974.

[2] A. Aho and J. Ullman, "Universality of Data Retrieval Languages," *Proc. 6th ACM Symposium on Principles of Programming Languages*, 1979.

[3] C. Apt, H. Blair, A. Walker, "Towards a Theory of Declarative Knowledge," *Foundations of Deductive Databases and Logic Programming*, J. Minker (ed.), Morgan Kaufmann, Los Altos, 1987.

[4] K. R. Apt and M. H. Van Emden, "Contributions to the theory of logic programming," *J. ACM*, vol. 29, no. 3, 1982, pp. 841–862.

[5] I. Balbin and K. Ramamohanarao, "A differential approach to query optimization in recursive deductive databases," TR-86/7, Dept. of Computer Science, University of Melbourne, 1986.

[6] F. Bancilhon, "Naive Evaluation of Recursively Defined Relations," *On Knowledge Base Management Systems—Integrating Database and AI Systems*, Brodie and Mylopoulos, Eds., Springer-Verlag, 1985.

[7] F. Bancilhon, D. Maier, Y. Sagiv and J. Ullman, "Magic Sets and Other Strange Ways to Implement Logic Programs," *Proc. 5th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems*, 1986.

[8] F. Bancilhon and R. Ramakrishnan, "An amateur's introduction to recursive query processing strategies," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Washington, D. C., May 1986, pp. 16–52.

[9] R. Bayer, "Query Evaluation and Recursion in Deductive Database Systems," Technical Report, Technische Universitaet, Munich, West Germany, January 1985.

[10] S. Ceri, G. Gottlob and L. Tanca, "What You Always Wanted to Know About Datalog (And Never Dared to Ask)," *Logic Programming and Databases*, Springer-Verlag 1990.

[11] A. K. Chandra and D. Harel, "Structure and complexity of relational queries," *J. Computer and System Sciences*, vol. 25, no. 1, 1982, pp. 99–128.

[12] C. L. Chang, "On Evaluation of Queries Containing Derived Relation in a Relation Database," in [15].

[13] W. Chen, M. Kifer and D. S. Warren, "Hilog: A First-Order Semantics for Higher-Order Logic Programming Constructs," *The Second International Workshop on Database Programming Languages*, Morgan-Kaufmann, June 1989.

[14] W. Chen and D. S. Warren, "C-logic for Complex Objects," *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, March 1989, pp. 369–378.

[15] H. Gallaire, J. Minker, J. M. Nicolas, *Advances in Database Theory*, vol. 1, Plenum Press, 1981.

[16] H. Gallaire, J. Minker and J. M. Nicolas, "Logic and databases: a deductive approach," *Computing Surveys*, vol. 16, no. 1, 1984, pp. 154–185.

[17] G. Gardarin, P. Valduriez, *Relational Databases and Knowledge Bases*, Addison-Wesley Publishing Company, New York, 1989.

[18] R. Gonzalez-Rubio, J. Rohmer and A. Bradier, "An overview of DDC: a delta driven computer," DSG/CRG/87007, Bull, Louveciennes, France, 1987.

[19] M. Kifer and E. Lozinskii, "Query Optimization in Logic Databases," Technical Report, SUNY at Stonybrook, June 1985.

[20] M. Kifer and J. Wu, "A Logic for Object-Oriented Logic Programming (Maier's O-logic Revisited)," *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, March 1989, pp. 379–393.

[21] J. W. Lloyd, *Foundations of Logic Programming*, 2nd extended edition, Springer-Verlag, New York, 1987.

[22] D. Maier, "A Logic for Objects," *Workshop on Foundations of Deductive Databases and Logic Programming*, Washington D. C., August 1986, pp. 6–26.

[23] D. McKay and S. Shapiro, "Using Active Connection Graphs for Reasoning with Recursive Rules," *Proc. IJCAI*, 1981.

[24] J. Minker, "Perspectives in deductive databases," Technical Report #1799, Computer Sciences Department, University of Maryland, 1987.

[25] S. Naqvi, "A Logic for Negation in Database Systems," Workshop on Deductive Databases, University of Maryland, 1986.

[26] J. F. Naughton, R. Ramakrishnan, Y. Sagiv and J. D. Ullman, "Argument Reduction by Factoring", *Proc. 15th Int. Conf. on Very Large Data Bases*, 1989, pp. 173–182.

[27] J. F. Naughton, R. Ramakrishnan, Y. Sagiv and J. D. Ullman, "Efficient evaluation of right-, left- and combined linear rules," *Proc. ACM SIGMOD Conf. on Management of Data*, 1989, pp. 235–242.

[28] T. Przymusinski, "On the Semantics of Stratified Deductive Databases," Workshop on Deductive Databases, University of Maryland, 1986.

[29] R. Ramakrishnan, "Magic Templates: A Spellbinding Approach to Logic Programs," *Journal of Logic Programming*, vol. 11, 1991, pp. 189–216

[30] R. Ramakrishnan, D. Srivastava and S. Sudarshan, "Rule Ordering in Bottom-Up Fixpoint Evaluation of Logic Programs," *IEEE Transactions on Knowledge and Data Engineering*, vol. 6, no. 4, August 1994, pp. 501–517.

[31] K. A. Ross, "On Negation in HiLog," *Journal of Logic Programming*, vol. 18, no. 1, January 1994, pp. 27–53.

[32] D. Sacca and C. Zaniolo, "Differential fixpoint methods and stratification of logic programs," *Proc. Third Intl. Conf. on Data and Knowledge Bases*, Jerusalem, Israel, 1988.

[33] J. D. Ullman, "Implementation of Logical Query Languages for Databases," ACM SIGMOD 1985, in ACM TODS, vol. 10, no. 3, September 1986.

[34] Jeffrey D. Ullman, *Principals of Database and Knowledge-Base Systems, Volume 1*, Computer Science Press, Rockville, MaryLand, 1988.

[35] Jeffrey D. Ullman, *Principals of Database and Knowledge-Base Systems, Volume 2*, Computer Science Press, Rockville, MaryLand, 1989.

[36] M. H. Van Emden and R. A. Kowalski, "The semantics of predicate logic as a programming language," *J. ACM*, vol. 23, no. 4, 1976, pp. 733–742.

[37] C. Zaniolo, "Safety and Compilation of Non-Recursive Horn Clauses," MCC Tech. Report D13-088-85.